# PROYECTO FIN DE MÁSTER EN INGENIERÍA DE COMPUTADORES

DEPARTAMIENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA
MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID


# OPTIMIZATION OF DYNAMIC MEMORY MANAGERS IN HIGH PERFORMANCE EMBEDDED SYSTEMS USING GRAMMATICAL EVOLUTION

Autor:
Rubén Gonzalo Ramiro

Directores:
José Ignacio Hidalgo Pérez
José Luis Risco-Martín

Curso académico 2008/2009

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: “Optimization of dynamic memory managers for high performance embedded systems using gramatical evolution”, realizado durante el curso académico 2008-2009 bajo la dirección de José Ignacio Hidalgo y José Luis Risco-Martín en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fdo. Rubén Gonzalo Ramiro

# AKNOWLEDGMENTS / AGRADECIMIENTOS


**E**n primer lugar me gustaría comenzar este espacio dando un sincero agradecimiento a Iñaki, probablemente sin sus recomendaciones y consejos no me habría embarcado en este proyecto. Tampoco hubiese sido posible llegar hasta aquí si no hubiese sido por el apoyo constante de Josele, quien ha estado ahí en todo momento durante este año, cuando he tenido una duda, un problema o me ha hecho falta, a pesar de lo pesado que he podido llegar a ser. También quiero mostrar mi agradecimiento a David, quien ha hecho posible que haya podido disfrutar durante seis meses de una estancia en IMEC en Bélgica y haya vivido una experiencia inolvidable, creciendo mucho como investigador, como trabajador y personalmente. De verdad que os estoy muy agradecido a los tres porque a pesar de que ha habido momentos malos, sobre todo en la lejanía donde todo se hace más difícil, siempre estuvisteis al otro lado del teléfono o del email para ayudarme e intentar solucionar los problemas que pudiese haber.

Por supuesto mi mayor agradecimiento va hacia mi familia e Irene, a los que siempre les toca apoyarme, comprenderme y finalmente sufrirme con todos los proyectos en los que me involucro. Gracias a ellos es más fácil sobrellevar los momentos de dudas e incertidumbre que en muchas ocasiones me asaltan y poder seguir adelante con más fuerza y entusiasmo.

No me gustaría dejar de lado a todos aquellos que me han ayudado en este año, tanto a nivel de investigación como personal. A mis amigos de toda la vida, que a pesar de pasar unas cuantas temporadas un poco lejos, al volver siempre están ahí. A todas aquellas personas que he conocido en Bélgica, tanto compañeros de IMEC como muchos Erasmus con los que me ha tocado compartir residencia, amistad y alguna que otra Stella.

Quiero agradecer a IMEC (Lovaina), a pesar de todos los malentendidos habidos, el haberme dado la oportunidad de realizar una estancia de seis meses allí. En especial me gustaría recordar a todos mis compañeros del grupo ARES, con los que he compartido la gran parte de mi tiempo allí. Me han ayudado mucho, hemos pasado buenos momentos y me han enseñado mucho.

Y para finalizar quiero dar mi agradecimiento a La Caixa, por haber depositado su confianza en mí de entre tantos otros candidatos y haberme seleccionado para disfrutar de una de las Becas para Estudios de Máster en España.

**F**irst of all I am very grateful to Iñaki, thanks to your advices finally I decided to do this master and I am very proud of it. I want to thank also to Josele, who has been always there. We have been a kind of team, he has been my adviser and he has helped me with all the doubts and problems that I have had during this research. I would like to thank also David, he has done possible my stay during six months in IMEC (Leuven), and it has been an unforgettable experience. I have grown like a researcher but also personally. I am very grateful to the three of you. You have been there when I have had problems, overall when I was abroad, and you have tried to make me feel the more comfortable I could.

Of course, my great gratitude goes to my family and Irene. You always understand me, support me and finally you have to suffer me with all the projects that I do. Thanks to you it is easier to overcame all the doubts and uncertainly moments that I have and go on stronger and with enthusiasm.

I would not like to forget all the people that have helped me along this year, at research and a personal level. My friends, they are always there, even when I spend some time abroad, when I come back I can be sure that they will be there. I would like to remember also all the friends that I have done during my Leuven life. They are researches at IMEC and also Erasmus student. They have had the common denominator. We have spent together really good times.

I want to thank IMEC, despite the initial misunderstanding, for let me to stay there during six months. It has been an unforgettable experience and I want to thank all my workmates and mainly the ones from the ARES group. We have shared a lot of laughs and nice moments and of course they have taught to me a lot of things.

Finally my last gratitude is for La Caixa. They have trust in me among a huge number of candidates and they gave me a grant: Becas para Estudios de Máster en España.

**To all of you, THANK YOU**
**A todos, MUCHAS GRACIAS**


***It is not the strongest of the species that survives, nor the most intelligent, but rather the one most adaptable to change.***
***No son las especies más fuertes ni las más inteligentes las que sobreviven, sino aquellas que mejor se adaptan al cambio.***

**Charles Darwin.**

# TABLE OF CONTENTS

# RESUMEN/SUMMARY


Los sistemas empotrados son pequeños dispositivos electrónicos con una o varias funciones dedicadas y que se encuentran integrados en un dispositivo más general. Hoy en día vivimos rodeados de este tipo de sistemas aunque en ocasiones no nos percatemos de ello porque forman parte de nuestra vida cotidiana (un coche suele incluir más de 200). Este trabajo se centra en los sistemas empotrados de altas prestaciones (móviles de última generación, pda's, reproductores portátiles, etc.). Estos dispositivos tienen que hacer frente a una serie de restricciones que anteriormente no eran un problema. Es necesario un alto rendimiento (para satisfacer las necesidades del usuario), a la vez que un consumo reducido de energía (para alargar la duración de las baterías) y un uso moderado de memoria (para abaratar el coste del dispositivo final).

Los sistemas empotrados de altas prestaciones son de naturaleza dinámica y tienen un alto grado de impredecibilidad. Para hacer frente a estas características, los sistemas se dotan de un mecanismo que gestiona los accesos a la memoria, el llamado gestor de memoria dinámica. En este trabajo se plantea un innovador flujo de diseño para diseñar gestores de memoria dinámica optimizados para el sistema en estudio. A partir de un estudio del perfil de las ejecuciones típicas del dispositivo, de forma automática y sin apenas intervención por parte del diseñador se crea un gestor para la memoria dinámica que optimiza la energía consumida, el rendimiento del sistema y el uso de memoria del mismo. Para la optimización se utilizan algoritmos de gramática evolutiva. Nuestros resultados experimentales muestran que se consigue una reducción del consumo de energía de hasta un 70%, una reducción de la memoria usada de hasta un 20% y un aumento del rendimiento de hasta un 56% en comparación con sistemas de propósito general utilizados en este tipo de sistemas. Estos resultados se consiguen reduciendo el tiempo necesario para diseñar el gestor hasta 25 veces respecto del tiempo necesario para el diseño de gestores en anteriores metodologías.

Todo el sistema desarrollado es paralelizable, pudiendo reducir aún más el tiempo de cómputo (hasta en un 25% más). Finalmente se ha realizado un estudio sobre la posibilidad de añadir fiabilidad al sistema haciendo uso del gestor de memoria, concluyendo que con las metodologías planteadas se puede reducir la probabilidad de fallos en memoria, a la vez que se mantienen o incluso mejoran las principales métricas para los sistemas empotrados.



**PALABRAS CLAVE**: Optimización, Computación Evolutiva, Gramáticas Evolutivas, Gestor de Memoria Dinámica, Diseño de Sistemas Empotrados, Paralelización, Fiabilidad



The embedded systems are specialized computer systems that are part of a larger system or machine. They are deployed by the billions each year in myriad applications and we can find it everywhere (a car usually has more than 200 embedded systems). This research work is focused in the high performance embedded systems (mobile phones, pda's, multimedia players, etc.) that in the last years had became to be widely used in our lives. The design of these systems is difficult because they must meet some constraints, mainly in performance (to offer a quick response to the user), in energy consumption (because of the battery) and in the final memory used (to reduce the final costs of the system).

The main problem of these systems is the dynamic nature and the unpredictability of the inputs, due to the high user interaction. These features make necessary the addition of extra mechanisms that manage the unpredictable memory accesses. These mechanisms are the dynamic memory managers. Usually adaptations of general purpose dynamic memory managers or ad-hoc dynamic memory managers have been used in the embedded systems, but or they do not meet all the constraints aforementioned or they require a really long design process. We present a new and innovative design flow in this research work. It automatically, i.e. without designer interaction, and in a fast way creates a dynamic memory manager that optimizes the system in energy consumption, in performance and in memory usage. To this end we use a profile of the typical applications run in the device and grammatical evolutionary optimization algorithms. Our experimental results show that using our methodologies it is possible to obtain improvements in performance of 56%, reductions in memory used of 20% and reductions in energy consumption of 70% in contrast to general purpose approaches. Furthermore, the computational design time necessary is 25 times less than previous methodologies.

Since the system is suitable to be parallelized, it is possible to reduce still more the time needed to design dynamic memory managers (in a 25% less time). Finally we have done an analysis to add reliability to the final system using the dynamic memory managers. The conclusions obtained are that it is possible to reduce the probability of errors in memory maintaining or even improving the main metrics, performance, energy consumption and memory usage.



**KEYWORDS**: Optimization, Evolutionary Computation, Grammatical Evolution, Dynamic Memory Managers, Embedded Systems Design, Parallelization, Reliability

# 1. INTRODUCTION

Around 1971 took place a big hit in the electronic era, the first microprocessor was designed. This microprocessor was the Intel 4004 produced by Intel, and this fact began to show that very small computers might be possible. Some years after this event, in 1981, the first personal computer appeared in the market (IBM PC model 5150). If we look at that personal computer and we compare with a personal computer that we have in our desktop, we could see that externally the differences are no so big, but if we go deep and we look at the specifications, the differences became really huge. In fact, the first personal computers were designed to run only a reduced set of applications. It means that they were produced for a specific purpose, but today's personal computers are produced for run a big and dynamic set of applications. They are called general purpose computers because they are used for a really big number of different functions. The improvement in the capabilities of the computers comes from the development in the integration technologies. In 1965 Gordon Moore (cofounder of Intel Corporation) predicted his famous Moore's Law [2]. This law formulates that since the invention of the integrated circuit in 1958 the number of transistors that can be placed inexpensively on an integrated circuit has increased exponentially, doubling approximately every two years. And he was not wrong because until now, this law has been met.

With the development of the personal computers, also smaller devices called embedded systems had been developed. These systems were originally created to execute one or two functions in other systems, and they became to be in everywhere. As example all the electrical appliances include at least one of these embedded systems. But as said before, there is an improvement in the integration technologies, and of course this improvement affect also to the embedded devices. They come from the really simple original devices to extreme complex

devices in which we can found thousand of functionalities. Nowadays we can divide the embedded systems into two categories [3].

First of all we have the traditional systems. They are the legacy of the original ones, as the integration technologies they can be smaller or more efficient, but they conserve the reduced functionality and the simplicity. These kind of embedded systems usually are cheap, have a massive production and are fully integrated in our society. As example of this kind of devices, we can refer to most of systems included inside a lot of electric appliances like fridges, washer, etc.

On the other hand, we have more recently kind of embedded systems that have arisen in the last years as the integration technologies improved. The new kind of embedded systems are complex devices, with a lot of functionality that are oriented to multimedia services offered to the final users. As example of this second kind of embedded systems, we can see the high performance mobile phones, handheld game consoles, etc.

This research work is centered in the second kind of embedded devices, the high performance ones. The reason is because these systems are really complex and they must meet some constraints. It is necessary to see in more detail the problems in the design of this kind of devices.

## 1.1 Problem definition

Modern multimedia embedded systems have a lot of functionalities and because of that they are able to run applications coming from desktop systems at the same time that they can run specific applications. As a result, one of the most important problems that system designers face today is the integration of a great amount of applications coming from a general-purpose domain into a highly constrained device [4] where power consumption is a crucial design priority both at the hardware and software levels because it affects the operating time, weight and size of the final system (i.e., through the battery). Therefore, it is required to minimize the power consumption while satisfying the memory storage requirements and access requests of the various embedded software applications that co-exist on the same hardware platform.

In the past, most implementations that were ported to these embedded platforms stayed mainly in the classic domain of signal processing and actively avoided algorithms that employ Dynamic Memory (DM). The concept of DM arises when we need a variable amount of memory that can only be determined during runtime. Recently, with the emerging market of new portable devices that integrate multiple services such as multimedia and wireless network

communications, the need to efficiently use DM in embedded low-power systems has arisen. New consumer applications (e.g. 3D video applications) are now mixed signal and control dominated. They must rely on DM for a very significant part of their functionality due to the inherent unpredictability of the input data, which heavily influences global performance and memory usage of the system. Designing them using static worst-case memory usage solutions would lead to a too high overhead in memory usage and power consumption for these systems [5]. In addition, power consumption has become a real issue in overall system design (both embedded and general-purpose) due to circuit reliability and packaging costs [6]. Thus, optimization in general (and especially for embedded systems) has three goals that cannot be seen independently: memory usage, power consumption and performance.

Since the DM subsystem heavily influences performance and is a very important source of power consumption and memory usage, exigible system-level implementation and evaluation mechanisms for these three factors must be available at an early stage of the design flow for embedded systems. The dynamic memory managers (DMMs) are mechanisms to handle allocation and deallocation of dynamic memory. Current implementations of DMMs can provide a reasonable level of performance for general purpose systems [7] . However, these implementations do not consider power consumption or other limitations of target embedded platforms where these DMMs must run on. Thus, these general-purpose DMMs implementations are never optimal for the final target platform and produce large power and performance penalties. Consequently, system designers currently face the need to manually optimize the implementations of the initial DMMs on a case-per-case basis. This has to happen without detailed profiling of which parts within the DMMs implementations (e.g. internal data structures or links between the memory blocks) are the most critical parts (e.g. in power consumption) for the system. Moreover, adding new implementations of (complex) custom DMMs often prove to be a very programming-intensive and error-prone task that consumes a very significant part of the time spent in system integration of DM management mechanisms (even if standardized languages such as C or C++ offer considerable support). The time spent in the design process is really important because of the aggressive technology industry. A delay in the launch of a new electronic device could bring really bad consequences in sales to the producer company [8].

Recently, a new high-level programming and profiling approach has been presented [9-11]. This methodology that will be explained in the next chapter is used to create efficient DMMs based on previous profiling of the applications that will be executed in the embedded system. They achieve a balanced result between memory footprint and memory accesses.

The problem with the previous methodologies is that they need to make several decisions to finally obtain the DMM. In this research work, these decisions have been eliminated and the process is done automatically, reducing the total time needed to create the final DMM.

## 1.2 Contributions

The main contributions that arise from this research work are the following:

a) The definition of a grammar that covers all the design space for DMMs. All kind of DMMs can be formed from this grammar.

b) The presentation of a novel DMM design flow based in some concepts of previous methodologies. It improves the behavior of the final DMMs obtaining better values for the main metrics in study, performance, energy consumption and memory usage. In addition the process is done automatically and faster than previous methodologies.

c) The parallelization of the methodology to obtain a reduction in the time to create new DMMs.

d) A study of the possibility of adding reliability at the DMM level. This study shows that it is possible to reduce the probability of having errors without negative effect in the main metrics: performance, memory usage and energy consumption.

## 1.3 Organization

This section explains the purpose and content of the chapters in this research work. The second chapter reviews all the methods needed to afford the optimization flow that is defined in this work. First in that chapter are presented the DMMs. In the second place, the optimization is explained, emphasizing the evolutionary optimization algorithms. In chapter 3 we present our optimization flow designed to reach optimal DMMs implementations in terms of performance, energy consumption and memory. In that chapter we also present the results obtained using our methodologies showing that we outperforms the results obtained by other well known DMMs like Lea or Kingsley allocators or even by a custom DMM obtained by heuristic methods in [10]. Chapter 4 presents the way to parallelize the methodology presented to obtain a more

reduced time to create the final DMMs. In the chapter 5 is shown a study about the possible modifications in our methodology to create a more reliable system. Finally chapter 6 presents the conclusions extracted from this research work and the future work that can be developed after this research work.

# 2. BACKGROUND

## 2.1 Dynamic memory managers

Nowadays most of the applications for the embedded systems are written in high-level languages like Java or C++. In this kind of languages there is the possibility of use dynamic memory. This means that a request of more memory (malloc() in C language) or a request to free memory (delete () in C language) can be done at any time during the application execution. Since this process is done at execution time, we need a program (algorithm) to manage (allocate and deallocate) this dynamic memory. Usually the OS or the programming languages have its own general-purpose Dynamic Memory Manager.

Currently the basis of an efficient DM management in a general-context are already well established. A survey of dynamic storage allocation was published in 1995 by Wilson [7], and since then it has been considered one of the main references in DMM design.

Conventional DM management basically consists of two separate tasks, namely allocation and de-allocation [7]. Allocation is the mechanism that searches for a block big enough to satisfy the request of a given application and de-allocation is the mechanism that returns this block to the available memory of the system in order to be reused later by another request.

So an allocator is an online algorithm, which must respond to request in strict sequence, immediately, and its decisions are irrevocable.

### 2.1.1 Fragmentation

The main problem with the DM allocators is that an application program may free blocks in any order, creating “holes”. If the holes are too small and numerous, they cannot be used to satisfy future request for larger blocks. This problem is the main problem in the management of the dynamic memory and is known as fragmentation (Figure 1). We can define fragmentation as the inability to reuse memory that is free.

The fragmentation can be divided in two categories [12]. The first one is called external fragmentation and it arises when free blocks of memory are available for allocation, but cannot be used to hold objects of the sizes actually requested by a program. That is usually because the free blocks are too small, and the program request larger objects like in the Figure 1. The other kind of fragmentation is called internal fragmentation and it arises when a large-enough free block is allocated to hold an object, but there is a poor fit because the block is larger than needed. So there is some space inside the block that is unused (the wasted space in the Figure 1).

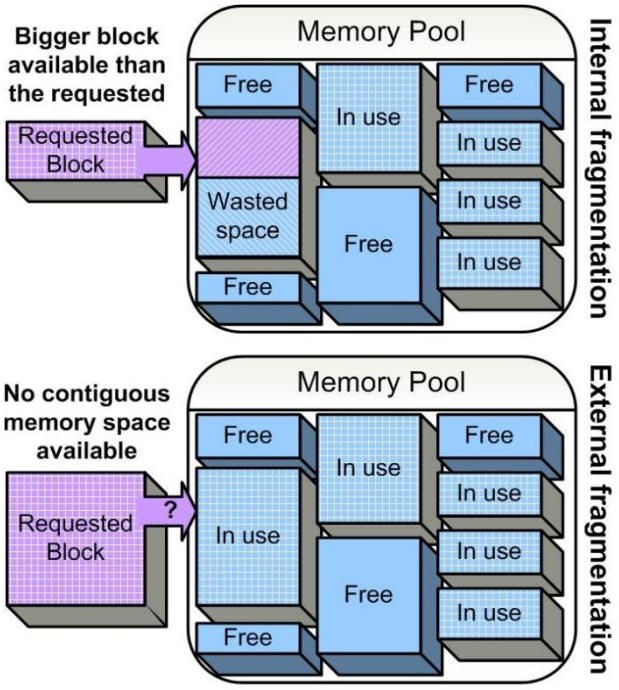


**Figure 1: Memory Fragmentation**

To combat the fragmentation there are two methods, splitting and coalescing. Split a block is the way to fight against internal fragmentation. The allocator will split blocks into multiple parts, allocating part of a block, and then regarding the remainder as a smaller free block in its own right. Coalesce block is the way to fight against external fragmentation. The allocator will coalesce two adjacent free blocks, combining them into larger blocks that can be used to satisfy request for larger objects.

It is really difficult to avoid the fragmentation because it is different depending on the application, i.e., the same mechanism to avoid fragmentation applied to two different applications will be different results in the final memory used.

Up to date, most of the DM allocators for embedded systems present the same structure and the general ideas that the general purpose dynamic memory managers developed several years ago. Some of these general purpose dynamic memory managers have been used to implement the managers of some operating systems. As an example we present two well-known DMMs: Doug's Lea Allocator and Kingsley Allocator.

*Doug's Lea Allocator:*

One of the most known, used and complex allocators is the Doug's Lea Allocator [11], [7] (also called Lea malloc or dlmalloc). The development of this allocator started in 1987, and since then it has been maintained and enhanced. It is used in the Linux based systems. The allocator is written in C and it provides implementations of the standard C routines malloc(), free(), and realloc(). The main feature of this allocator is that it is among the fastest while also being among the most space-conserving, portable and tunable. These reasons make Lea Allocator a good general-purpose allocator for malloc-intensive programs.

The structure to maintain the blocks management is very complex as it can be observed in the Figure 2. It has a set of lists where insert the free blocks. In addition, to limit the number of lists that will be necessary to cover to find the correct list in a request, the field binmap in the header of the lists has the information about the ranges of the block sizes for each list. There are 96 FIFO doubly linked lists to manage the small chunks (is the term that they use to refer to a block inside the Lea Allocator): 64 lists for 8 bytes chunk and 32 for 64 bytes chunk. To manage the medium chunks there are 31 FIFO doubly linked lists: 16 for 512 bytes chunks, 8 for 4KB chunks and 7 for 32 KB. To manage the big chunks, between 256KB and 1MB there is one FIFO doubly linked list with the policy of first fit. If there are requests of more than 1MB, then the allocator request an

additional region of memory using the virtual memory support of the OS (mmap() function). When this memory is freed it is returned directly to the system.

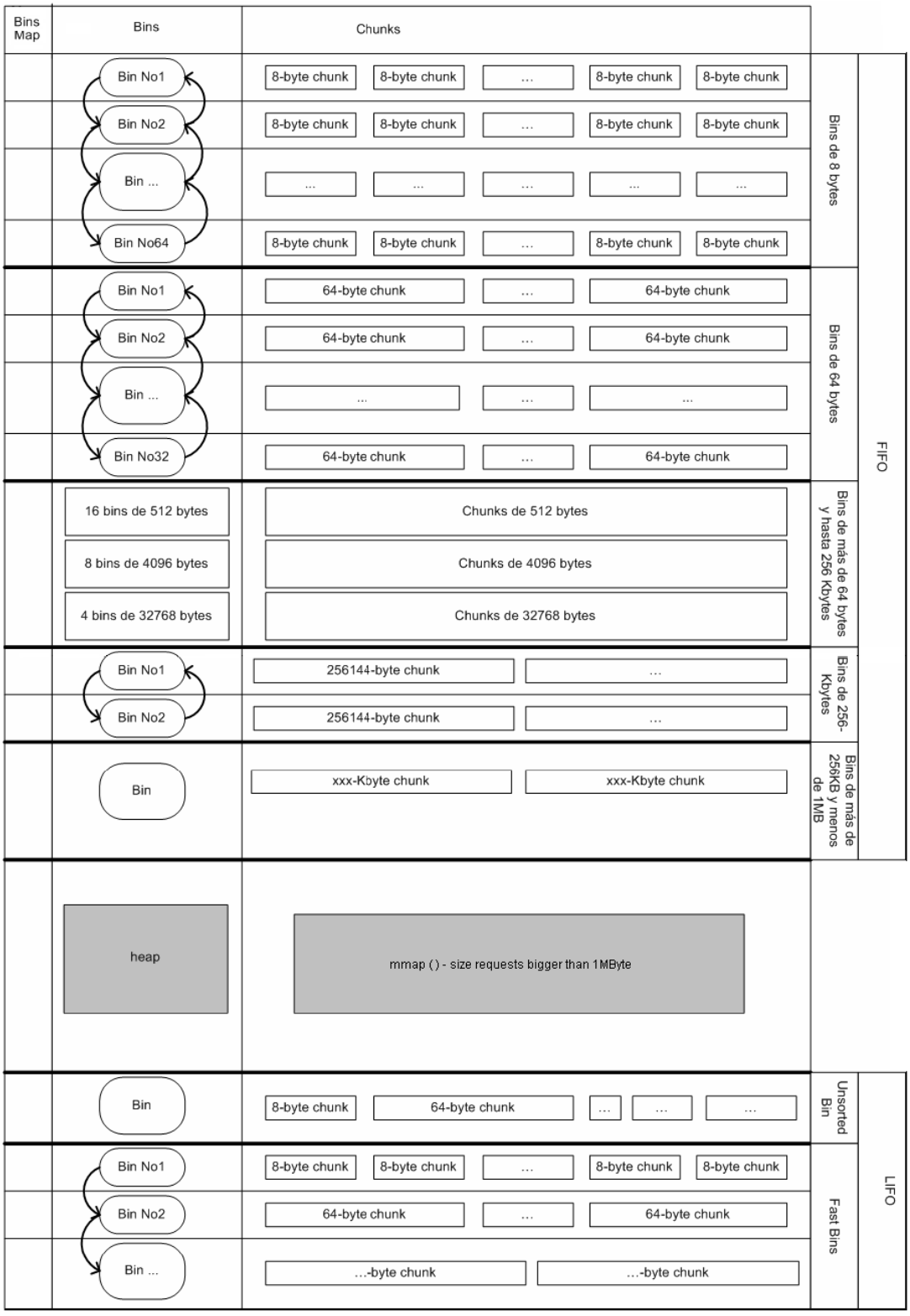


**Figure 2. Internal Architecture. Lea Allocator**

The mechanism of coalescing and splitting is used when a size requested does not fit exactly with any of the lists. In the case of the small and medium sizes chunks, there is an additional list called unsorted bins that stores all the chunks that have sizes different from the ones of the other bins. These chunks are formed after coalescing or splitting. For the big chunks, as the list can store a block with any size from the range of sizes there are no extra lists. Finally there exists a reduced set of lists that include the small and medium chucks recently freed (fastbins). This list is maintained because it helps when successive request of blocks of the same size are done. In these two last cases the policy used by the lists is LIFO.

All the chunks include an additional header. This header is different if the chunk is free or is in use, using four or eight bytes respectively. The minimum size of the structure (header + info) is 16 bytes. Chunks are maintained using a `boundary tag' method as originally described by Knuth [7]. In the Figure 3 is shown the chunk when it is free and when it is in use. In the case of a free chunk, the fields fd and bk are references to the next and the previous chunk respectively, creating a doubly linked structure. Moreover the header contains the size of the block and the size of the previous block. In the case of a used chunk, the fields bk and fd are available to use as part of the block.

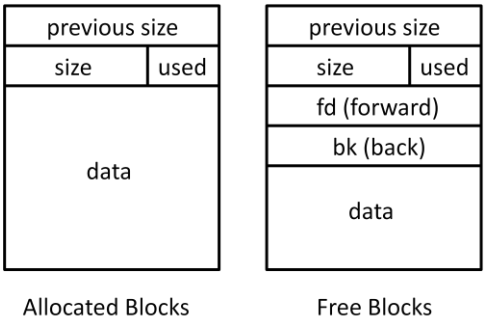


**Figure 3. Chunks Structure. Lea Allocator**

*Kingsley Allocator:*

Kingsley [7], [13] allocator was developed in 1982. It is still used and is the fundamental piece for the allocators included in the OS Windows CE, Windows NT, Windows Pocket PC and Unix FreeBSD. It is one of the fastest general-purpose allocators although it is among the worst in terms of fragmentation.

The Kingsley allocator is one example of a segregated fits allocator. Segregated fits allocators divide objects into a number of size classes, which are ranges of object sizes.

Memory requests for a given size are satisfied directly from the "bin" corresponding to the requested size class. The heap returns deallocated memory to the appropriate bin.

The Kingsley allocator is a power-of-two segregated fits allocator: all allocation requests are rounded up to the next power of two (and this data is really fast to calculate). This rounding can lead to severe internal fragmentation (wasted space inside allocated objects), because in the worst case, it allocates twice as much memory as requested. Once an object is allocated for a given size, it can never be reused for another size: the allocator performs no splitting or coalescing. This allocator usually has a big memory footprint because it always maintains a structure in memory even when there are no free blocks. In the Figure 4 is shown the internal architecture of this allocator that includes 29 bins of fixed size blocks in each one. In the Figure 4 are drown the blocks in each bin or size class, but it is only for a better understanding. These blocks are only if the bin contains free blocks.

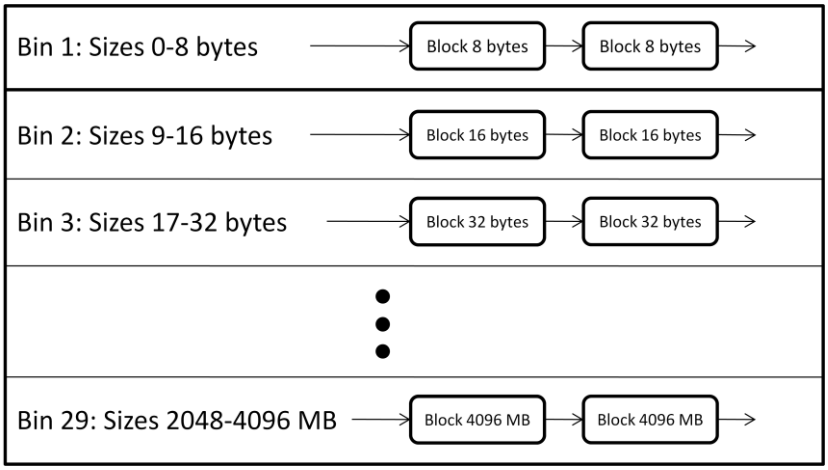


**Figure 4. Internal Architecture. Kingsley Allocator**

But the use of these general purpose DMMs in the embedded systems have the consequence that some of the constraints are not satisfied, e.g. the Lea Allocator can obtain a reduced memory footprint for an application, but at the same time the time to execute the application could exceed the expected time. To afford this fact, some methodologies have arisen to create DMMs specific for each system and application, trying to meet all the constraints. These DMMs are called custom DMMs (cDMMs).

### 2.1.2 Custom Dynamic Memory Managers

Normally, to create custom DMMs some information is needed about the behavior of the set of applications that will be run in the system. Among the relevant information extracted from the applications, we can find the sizes of the data requested, when this data is requested, how many times we access to the same data, which data are alive at the same time, etc. From this information, it is easier to suppose which kind of DMM will obtain the best result in terms of fragmentation and try to get the best one, but the process is not so easy. The process of examining the data available in the application and collecting statistics and information about that data is called profiling [14]. There are previous works that using a profiling phase to get a better performance [15], or decrease the impact of other metrics like for example the memory usage.

With the use of the profiling report, more complete research on cDMMs that take application-specific behavior into account to improve performance has appeared.

Vmalloc [16] lets the programmer define multiple regions (distinct heaps) with different disciplines for memory de/allocation for each. The programmer performs customization by supplying user-defined functions and structs that manage memory. By chaining these together, vmalloc does provide the possibility of composing heaps.

Berger et al [11], proposed an infrastructure of C++ layers that can be used to improve performance of general-purpose managers and also can be reused in other implementations. It is possible to construct since general purpose managers to custom managers. However, this approach lacks of the required formalization to consistently design and profile cDMMs and they cannot extend the improvement in performance to other metrics like energy consumption (that is very important in the embedded systems)

Finally, there is a set of works [10], [17], [9] y [18] in which their authors expose a new methodology to implement dynamic memory managers through an exhaustive exploration. They propose a complete taxonomy of the dynamic memory managers. They can create custom allocators based in some metrics like the reduced use of memory [10] or reduce the power consumption [9] or reduce the memory accesses [17] or even balanced some of these metrics [18] to get a better allocator. However they need to make a lot of decisions based in the profiling report to reduce the search space for DMMs and the time spent to create new DMMs is too long.

### 2.1.3 Dynamic Memory Managers Design Space

In spite of the fact that there is a lot of literature available about the memory managers [7], there is not any research previous to the work that we are going to review [3], [10] in the following paragraphs that have done a complete decisions space in the memory allocators to have a complete exploration in all the decisions that can be taken.

We have talked before about the main problem within the dynamic memory allocators, that is fragmentation. To support all the mechanisms to avoid fragmentation, additional data structures should be built to keep track of all the free and used blocks, and the defragmentation mechanisms. As a result, to create an efficient DM manager, there are a lot of decisions that can be taken.

Atienza et al. [10] have classified all the important design options that constitute the design space of dynamic memory management in different orthogonal decision trees (Figure 5). Orthogonal means that any decision in any tree can be combined with any decision in another tree, and the result should be a potentially valid combination, thus covering the whole possible design space. Then, the relevance of a certain solution in each concrete system depends on its design constraints, which implies that some solutions in each design may not meet all timing and cost constraints for that concrete system. Moreover, the decisions in the different orthogonal trees can be ordered in such a way that traversing the trees can be done without iterations, as long as the appropriate constraints are propagated from one decision level to all subsequent levels.

Basically, when one decision has been taken in every tree, one custom dynamic memory manager is defined for a specific dynamic memory behavior pattern. In this way, it is possible to recreate any available general purpose dynamic memory manager or create new highly specialized dynamic memory managers.

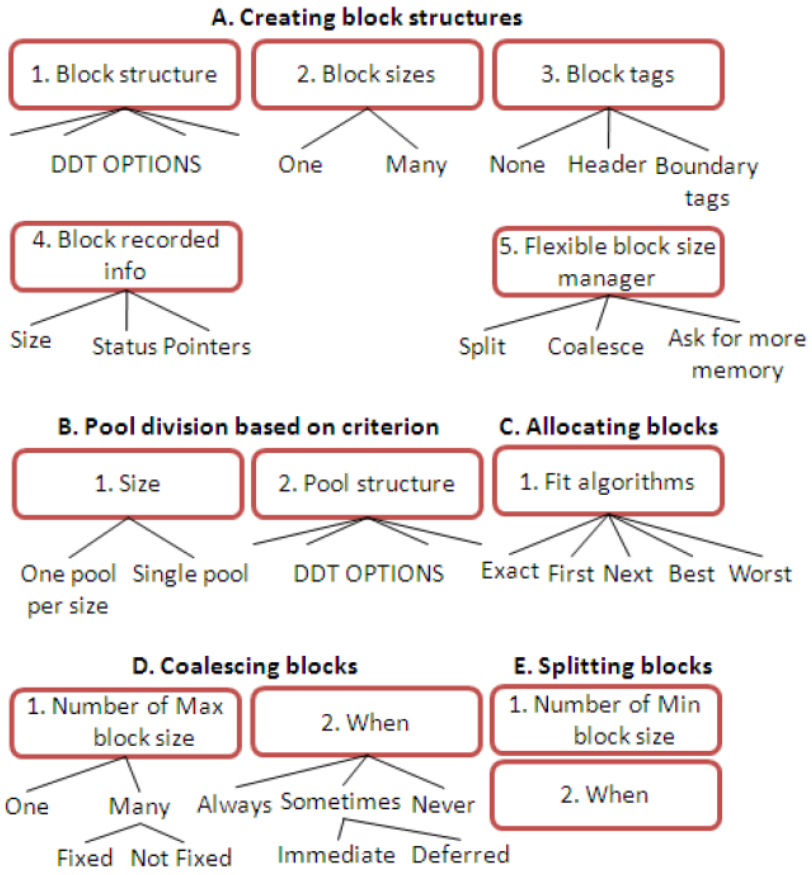


**Figure 5. DMM Design Space**

Then, authors in [10] have grouped the trees in categories according to the different main parts that can be distinguished in DM management [7]. There are five main categories (see Figure 5) and each one has its own trees. A brief description of all of them is the following:

A. Creating block structures. It deals with the creation and the later use of the data structures that the dynamic memory manager needs to satisfy the memory requests. The Block structure tree specifies the different blocks of the system and their internal control structures. The Block sizes tree refers to the different sizes of basic blocks available for DM management, which may be fixed or not. The Block tags and the Block recorded info trees specify the extra fields needed inside the block to store information used by the dynamic memory manager. And the Flexible block size manager tree decides if the splitting and coalescing mechanisms are activated or extra memory is requested from the system. This depends on the availability of the size of the memory block requested.

B. Pool division based on. It deals with the number of pools (or memory regions) present in the DMM and the reasons why they are created. The Size tree means

that pools can exist either containing internally blocks of several sizes or they can be divided so that one pool exists per different block size. The Pool structure tree specifies the global control structure for the different pools of the system.

C. Allocating blocks. It deals with the actual actions required in dynamic memory management to satisfy the memory requests and couple them with a free memory block. Here it is included all the important choices available in order to choose a block from a list of free blocks [7]. Note that a Deallocating blocks category with the same trees as this category could be created, but is not included to avoid adding complexity unnecessarily to the dynamic memory management design space. These two categories are so tightly linked together regarding memory footprint of the final solution that the decisions taken in one must be followed in the other one. Thus, the Deallocating blocks category is completely determined after selecting the options of this Allocating block category.

D. Coalescing blocks. It is related to the actions executed by the DM managers to ensure a low percentage of external memory fragmentation, namely merging two smaller blocks into a larger one. The Number of max block size tree defines the new block sizes that are allowed after coalescing two different adjacent blocks. The When tree defines how often coalescing should be performed.

E. Splitting blocks. It refers to the actions executed by the DM managers to ensure a low percentage of internal memory fragmentation, namely splitting one larger block into two smaller ones. The Number of min block size tree defines the new block sizes that are allowed after splitting a block into smaller ones. The When tree defines how often splitting should be performed (these trees are not presented in full detail in Figure 5, because the options are the same as in the two trees of the Coalescing category).

The decision trees described above are orthogonal, but not independent, this means that there are some interdependences that must be defined before continue. Therefore, the selection of certain leaves in some trees heavily affects the coherent decisions in the others (i.e., interdependencies) when a certain DMM is designed. The interdependencies can be classified in two main groups:

- Leaves or Trees that Obstruct the Use of Others in the New Design Space. These interdependencies appear due to the existence of opposite leaves and trees in the design space. First, inside the Creating block structures category, if the none leaf from the Block tags tree is selected, then the Block recorded info tree cannot be used. Clearly, there would be no memory space to store the recorded info inside

the block. Second, the one leaf from the Block sizes tree excludes the use of the Size tree in the Pool division based on criterion category. In a similar way, the one leaf from the Block sizes tree, excludes the use of the Flexible block size manager tree in the Creating block structures category. This occurs because the one block size leaf does not allow us to define any new block size.

- Leaves or Trees that Limit the Use of Others in the New Design Space. These interdependencies exist since the leaves have to be combined to create consistent whole DM schemes. For example, the coalescing and splitting mechanisms are quite related and the decisions in one category have to find equivalent ones in the other one. First, the Flexible block size manager tree heavily influences all the trees inside the Coalescing Blocks and the Splitting Blocks categories. Thus, according to the selected leaf for a certain atomic DMM (i.e., the split or coalesce leaf), the DMM has to select some leaves of the trees involved in those decisions or not. Second, the decision taken in the Pool structure tree significantly affects the whole Pool division based on criterion category. This happens because some data structures limit or do not allow the pool to be divided in the complex ways that the criteria of this category suggest. Third, the Block structures tree inside the Creating block structures category strongly influences the decision in the Block tags tree of the same category because certain data structures require extra fields for their maintenance. For example, single-linked lists require a next field and a list where several blocks sizes are allowed has to include a header field with the size of each free block inside[19]. Finally, the respective When trees from the Splitting and Coalescing Blocks categories are linked together because they are very tightly related to each other and a different decision in each of these two trees does not seems to provide any kind of benefit to the final solution.

Atienza et al. [10] propose that to create an efficient DMM that meet the constraints, an order in the exploration of the design space must be follow, depending on what is the metric that to optimize. To reach this conclusion, the authors have studied which are the influence factors for each of the relevant metrics (memory use, memory accesses and power consumption). In the next paragraphs these factors will be explained because they are interesting to understand some results and the dependences between metrics [3].

1) Regarding the total memory used by the system (includes the memory used by the DMM for the maintenance and also the memory used to allocate blocks), the influence factors are the organization overhead and the memory fragmentation.

a. The Organization overhead is the overhead produced by the assisting fields and data structures, necessary for each block and pool respectively. This organization is essential to allocate, deallocate and use the memory blocks inside the pools, and depends on:

   i. The fields (e.g., headers, footers, etc.) inside the memory blocks which are used to store data regarding the specific block and are usually a few bytes long. This is controlled by category A (Creating block structures) in the design space.

   ii. The assisting data structures provide the infrastructure to organize the pool and to characterize its behavior. They can be used to prevent fragmentation by forcing the blocks to reserve memory according to their size without having to split and coalesce unnecessarily. The use of these data structures is controlled by category B (Pool division based on criterion).

b. The Fragmentation memory waste is caused by the internal and external fragmentation, discussed earlier, which depends on the following:

   i. The internal fragmentation is mostly remedied by category E (Splitting blocks). It mostly affects to small data structures.

   ii. The external fragmentation is mostly remedied by category D (Coalescing blocks). It mostly affects to big data requests.

2) The number of memory accesses depends basically in the location of the block and if we use the fragmentation or not.

a. Localization of a block. First of all it is necessary to find the region in which the block will be, and then is necessary to find the block inside the structure that contains the blocks. The first step is influenced by the decision taken by the category B (pool division based on criterion) and the second step is influenced by the decision taken by the category A (creating block structures - block structure). Here there is a trade-off between number of regions and number of blocks in each region. If there are a lot of regions, the first step will have more memory accesses, but if we decide less regions, then we will have more blocks in each region and find a block inside a region will have more memory accesses.

b. Elimination of the fragmentation. The mechanism of coalescing and splitting entail a big number of memory accesses we have to look for the next or the previous block, or create new blocks. These decisions are taken by the categories E (splitting blocks) and D (coalescing blocks) depending on if it is internal or external fragmentation.

3) The power consumption of the memory system must be analyzed by the combination and exploration of the factors of influence of the previous two metrics, number of memory accesses and memory use. These two metrics are conflicting and it is necessary to make a trade-off between them. In [3] the author reaches the conclusion that the number of memory accesses has more impact in the power consumption than the memory use.

Taking in account this information and doing some experiments, in [3] the author reaches the conclusion that the most important element in the search space is the predominant block size, then depending if the most of the blocks are big, small or medium, he has classified other decisions to make depending on what is the metric to optimize.

Once the decisions to create a DMM have been manually made, it is necessary to implement the final DMM. To this end, some research works [10, 11] present a C++ library that cover all the DMM design space defined in this chapter. This library is described in the next section. In the current research work, we use this DMM search space as basis for our system. But we avoid the decisions to create an order in the exploration making the process automatic. We have created a tool called grammar tool that create a grammar that cover all the DMMs design space. Then this grammar is used to generate DMMs but without the designer interaction.

### 2.1.4 Supporting the DMM Design Space in C++

Atienza et al. [9] and Berger et al. [11] have developed a C++ library based on abstract classes and templates [20] that covers all the possible decisions in the DMM design space above mentioned. It enables the construction of the final global custom DM manager implementation in a simple way via composition of C++ layers. They have created several modules representing parts of a DM manager and it is easy to divide all the modules into some categories according to the DMM components:

- OS interfaces. Is a connector between the DMM and the OS. It includes the primitives to some different OS.

- Selectors. This layer indicates what selection policy is going to be used. It includes several policies like bestfit, worstfit, firstfit, etc.
- Block headers. Define all the options in the additional structures for the maintenance of a block. There are different headers depending on the fields that they have, e.g. without field (emptyHeader), with a field for the size (sizeHeader) or with extra fields for maintenance of lists (leaHeader)
- Coalescing block mechanisms. This level includes all the layers that let the DMM to include the coalescing mechanism (coalesceHeap, coalesceableHeap).
- Splitting block mechanisms. This level includes all the layers that let the DMM to include the splitting mechanism (slopHeap, thresholdHeap).
- Block management structures. It contains all the needed mechanism to support the management of the blocks, like lists of free blocks (fifoSL, lifoDL, etc.)
- Regions of blocks of DM. This level defines how the division of blocks of the memory is done. It can be done by sizes, by order, or by other criteria selected by the user(segHeap, selectMmapHeap, heapList)

To create a completely cDMM, in which the designer take all the decisions about the final DMM, the authors of [10] and [11] have created the basic interface for this purpose, named heapList. Each DMM is formed by a set of atomic DMMs, and each atomic DMM is defined by this heapList. The class prototype is as follow (Figure 6):

```
template<class Heap, class AllSel, class FreeSel, class Tail>
class HeapList {
    …
    inline void* malloc (size_t sz) {…}
    inline void free (void* ptr) {…}
}
```

**Figure 6. HeapList Interface**

Where:

- Heap is the data structure of the atomic DMM designed for a certain region of memory. It should include the type of data structure and policies for blocks sorting and selection that are used in that manager.
- AllSel includes the set of conditions determining the range of block sizes that will be attended by this atomic DMM. If there are several atomic DMMs with the same range, every memory request is attended in descending order as the atomic DMMs are created in the code, in such a way that the last atomic DMM attends requests when there are no free blocks on the previous atomic DMMs.
- FreeSel defines the set of rules determining the range of block sizes that are returned (freed) by this atomic DMM. Using this parameter, block migration policies between different atomic DMMs can be defined.
- Tail is the next atomic DMM in the global manager structure. It is a recursive link. If there are no more atomic DMMs, it represents the interface used by the Operating System (OS) to de/allocate memory (sbrk(), mmap(), malloc(), etc.).

Using the layer composition, it is possible to create complex DMMs, as an example, we can see the Figure 7, where there is a cDMM implemented using the C++ library described above. This DMM is the union of three atomic DMMs, which means that each manager manages a different region of memory that is selected depending on the size of the block requested. In this case, the blocks that has a size of 40 bytes will be attended by the first atomic manager (as for allocation as for deallocation). This region has a structure to manage the free blocks, that is a singled linked list with a policy of reubication FIFO (first in, first out) and the policy of extraction of Fixed, which means that always take the first block from the list. The next atomic manager will attend the blocks of 80 bytes with the same data structure as the previous explained to manage the free blocks. Finally, for the rest of the blocks that has different sizes, will be the third manager who attends the requests. In this case, the data structure used to manage the blocks is a single linked list with FIFO and best fit (take the best block from the list) policies. The last heap is the interface with the operating system. We reserved a region of memory of 2048 KB through a call to sbrk.

```
typedef SingletonHeap <
        HeapList<
                FIFOSLFixedListHeap<SizeHeader>,
                SizeSelector<40>,
                SizeSelector<40>,
                HeapList<
                        <FIFOSLFixedListHeap<SizeHeader>,
                        SizeSelector<80>,
                        SizeSelector<80>,
                        HeapList<
                                FIFOSLBestFitHeap<SizeHeader>,
                                TrueSelector,
                                TrueSelector,
                                FixedHeap<
                                        SbrkHeap<EmptyHeader>,
                                        2048,
                                        SizeHeader
                                >
                        >
                >
        >
> GlobalHeap;
```

**Figure 7. Example cDMM created with the library**

The creation of new cDMMs using this library is easy, fast and intuitive even for the more complex DMMs. It is possible to recreate any of the general purpose DMMs (e.g. Lea or Kingsley Allocators can be implemented using this library). Other feature of this library is the flexibility to modify existing DMMs, as easy as change the lines of the features that we want to change.

In this research work, this library has been extended to add a simulation working mode. The purpose of this mode is to change the traditional way to evaluate DMMs. Previous to this work, the evaluation of DMMs was done recompiling the application including the created DMM and re-executing it. In addition if the application requires the user interaction, someone must be there to interact. With the new and innovative simulation mode, to evaluate a DMM is not needed to recompile the code, automatically the behavior of the DMM is simulated with the information from a profiling report. Some information is collected in this simulation, and based in this information a measure of how good is the DMM is given. With this new approach it is possible to evaluate a huge number of DMMs in a short period of time and make a selection among all the DMMs evaluated.

At this point a general idea about the DMMs, the library to create it in an efficient way and some other concepts is given. As the objective of this research work is to obtain optimal

DMMs, the next step is show an introduction to optimization and to some optimization algorithms used in the current research work.

# 2.2 Optimization

## 2.2.1 Introduction

The goal of optimization is to find the best possible elements $x^*$ from a set $X$ according to a set of criteria F = {$f_1$, $f_2$, .., $f_n$}, where each $f_i$ is an objective function [1]. The objective functions give us a value that shows the quality of the evaluated elements.

So we can state that the aim of the optimization is to find an optimum, but the idea of optimum differs if we treat with simple or multiple objective optimizations.

If we have only one criterion f, then we are treating with single objective optimization. In this kind of optimization, an optimum is either its maximum or minimum, depending on what we are looking for.

Some definitions must be given to understand the simple objective optimization (see Figure 8 to understand it graphically):

- Local Maximum. A local maximum $x_l \in X$ of one objective function $f: X \rightarrow \boldsymbol{R}$ is an input element with $f(x_l) \geq f(x)$ for all $x$ neighboring $x_l$.
- Local Minimum. A local minimum $x_l \in X$ of one objective function $f: X \rightarrow \boldsymbol{R}$ is an input element with $f(x_l) \leq f(x)$ for all $x$ neighboring $x_l$.
- Local Optimum. A local optimum $x_l \in X$ of one objective function $f: X \rightarrow \boldsymbol{R}$ is either a local maximum or a local minimum.
- Global Maximum. A global maximum $x_l \in X$ of one objective function $f: X \rightarrow \boldsymbol{R}$ is an input element with $f(x_l) \geq f(x)\ \forall x \in X$.
- Global Minimum. A global minimum $x_l \in X$ of one objective function $f: X \rightarrow \boldsymbol{R}$ is an input element with $f(x_l) \leq f(x)\ \forall x \in X$.

- Global Optimum. A global optimum $x_l \in X$ of one objective function $f: X \to \boldsymbol{R}$ is either a global maximum or a global minimum.

There are normally multiple, often even infinity many optimal solutions. The optimal set is the set that contains all optimal elements. Since the memory of our computers is limited, we can find only a finite (sub-) set of them. We thus distinguish between the global optimal set and the set of seemingly optimal elements that an optimizer returns.

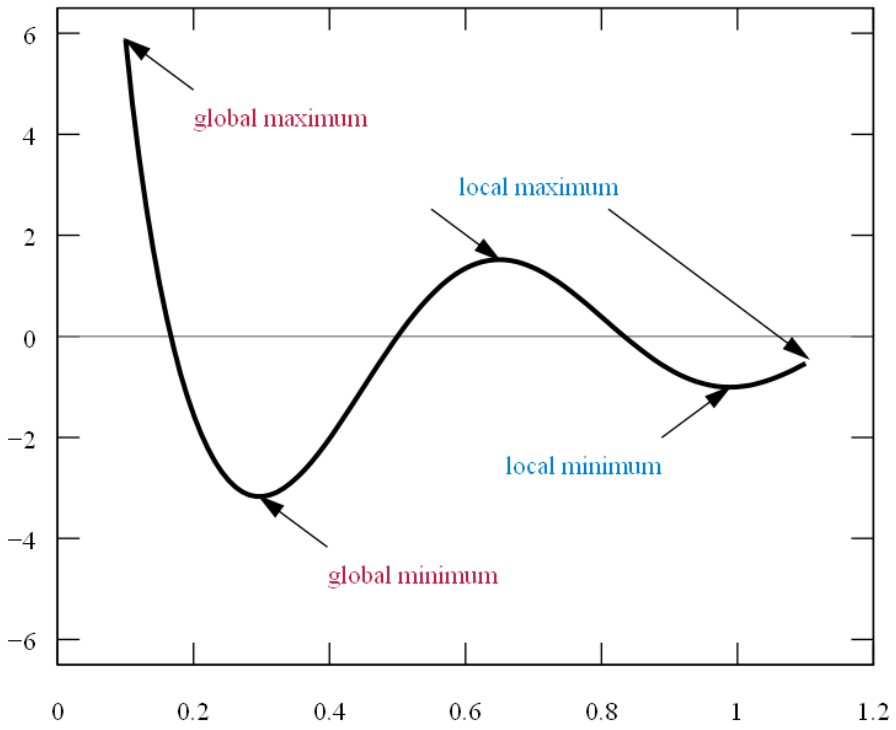


**Figure 8. Global and Local Optimus**

### 2.2.1.1 Heuristic and Metaheuristic

When an algorithm to optimize a problem is created, it could be deterministic or probabilistic. The main difference between them is that a deterministic algorithm always gets the same outputs for the same inputs, and in the probabilistic algorithms there is always a component of probability. Thus, for the same inputs, the outputs could be different. The deterministic algorithms are valid when a clear relation between the characteristics of the possible solutions and their utility for a given problem exists or the dimensionality of the search space is not very high (State Space Search or Branch and Bound search) but otherwise is necessary to use

probabilistic algorithms. In the process to find the optimal solution, is needed a heuristic or a metaheuristic to guide the algorithm in the path to the best solution.

- Heuristic. Is a part of an optimization algorithm that uses the information currently gathered by the algorithm to help to decide which solution candidate should be tested next or how the next individual should be produced.
- Metaheuristic. Is a method for solving very general class of problems. It combines objective functions or heuristics in an abstract and hopefully efficient way, usually without utilizing deeper insight into their structure.

#### 2.2.1.2 Multi-Objective Optimization Problems

Most optimization problems in the real world cannot be solved with only one objective function (single objective optimization), are the called multi-objective optimization problems (MOOP). We can find some of them without problems, e.g., in our case, in contrast to the last example, we need a DMM with a the less execution time possible, but at the same time we need low energy consumption, and these two objectives are not compatible. So it is needed a trade-off between the objectives. In Figure 9 the red line shows the decisions that can be taken compromising one objective by the other. To solve this kind of problems there are several ways to afford it, let's see some of them.

The first approach to solve the MOOP is to combine all the objectives in one simple scalar value (It could be seen also as a single objective optimization problem). This technique is called aggregating functions and is the balanced sum of all the functions:

$$g(x) = \sum_{i=1}^{n} w_i f_i(x)$$

Where $w_i > 0$ are the weighting coefficients representing the relative importance of the i objective function in the problem.

This way of deal with multi-objective optimization could have one limitation when the functions have different slope. Could be that one of the functions is dominant over the others so the weight has no effect (This not always happen). Other limitation is the added difficult of calculate the weights. [21].

A second approach to solve the MOOP is the Pareto optimality. This is a concept extracted from the neoclassical economics theory and it has a wide range of applications in the game theory, in the engineering and in the social sciences. The Pareto optimality defines a

frontier of solutions that can be reached taken some decisions that compromise one objective by others in an optimal manner. The red line in Figure 9 is the Pareto frontier for the optimization of the two objectives: execution time and energy consumption. From this front, a decision maker can finally choose the configuration that, in his opinion, suite best. To understand better the concept we can see some definitions [22] [1] [23]:

- Pareto Dominance: An element $x_1$ dominates (is preferred to an) element $x_2$ ($x_1 \prec x_2$) if $x_1$ is better than $x_2$ in at least one objective function and not worse with respect to all other objectives. In Figure 9 the element $x_1$ dominates the element $x_3$. Based on the set F of objective functions f, we can write:

$$x_1 \prec x_2 \Leftrightarrow \forall i: 0 < i \leq n \Rightarrow w_i f_i(x_1) \leq w_i f_i(x_2) \ \wedge$$

$$\exists j: 0 < j \leq n \Rightarrow w_j f_j(x_1) < w_j f_j(x_2)$$

$$w_i = \begin{cases} 1 \ if \ f_i \ should\ be\ minimized \\ -1 \ if \ f_i \ should\ be\ maximized \end{cases}$$

- Pareto Optimal: An element $\boldsymbol{x}^* \in X$ is Pareto optimal (and hence, part of the optimal set) if it is not dominated by any other element in the problem space. In Figure 9 the element $x_1$ and the element $x_2$ are Pareto optimal elements
- Pareto frontier or Pareto set: Is the set of choices that are Pareto optimal. Figure 9 shows the Pareto frontier or Pareto set with a red line.

$$\boldsymbol{x}^* \in \boldsymbol{X}^* \Leftrightarrow \nexists x \in X: x \prec \boldsymbol{x}^*$$

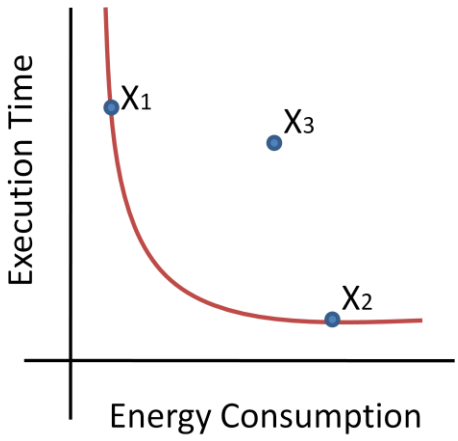


**Figure 9. Pareto Dominance**

Nowadays this field is in constant research, so there are a lot of different approaches to afford the MOOP. In our case we use a kind of algorithms called evolutionary algorithms (EA) that are explained in the next section. To obtain a measure of the quality of a DMM, in this work we use a balanced sum of the important metrics in the embedded systems, performance, energy consumption and memory usage. In a future extension of this work the balanced sum could be replaced by Pareto optimality.

## 2.2.2 Evolutionary algorithms

### 2.2.2.1 Introduction

In 1859 Charles Darwin published his book "*On the origin of Species*" [24] in which he argued that all the species share a "common ancestor" and all the changes that we can observe in the species is the result of a slow process of development in each of the species. This process arises from the heredity from the parent or parents and the environment. So only the individuals that adapt better to their environment will be the ones that survive and could be parents in the next generation.

Some years later, Gregor Mendel, contributed to the evolution theories with his famous three laws of Mendel [25]. These laws were discovered treating with pea plants. The laws explain how peas with the same external appearance can be crossed and in the next generation the resulting peas could be different.

The Neo-Darwinist theory and the modern evolutionary synthesis merge the previous explained theories. It establishes that the life in our planet can be described in terms of these four processes: reproduction, mutation, competition and selection.

These ideas were used and in the late 1950's came up a new form to deal with the optimization problems based exactly in the nature evolution, the idea is that if there is a set of candidate solutions for an optimization problem, should be possible to find better solutions combining the good properties of the previous candidate solutions. The name given for these algorithms was Evolutionary Computation [26].

There is a complete terminology related to the EA, so the best thing is to start explaining all the concepts to see later how these algorithms work.

### 2.2.2.2 Representation and main Elements

The representation links the "real world" with the "world of the evolutionary algorithms". If we look into the real world, we find that a possible solution in this original context is called phenotype. If we translate this solution to the world of the evolutionary algorithms to work with it, then we have the genotype. As example, in an optimization problem of integers, the phenotype (data extracted from the real world) 5 could be translate into the genotype 101, this genotype is the data that codify the solution for treat it in the algorithm. A solution is obtained by decoding the best genotype after termination.

In addition, the components can be named in different ways: in the original context, phenotype, candidate solution and individual are used to denote points of the space of possible solutions. On the other side, genotype, chromosome, and individual can be used for points in the space where the evolutionary algorithm will actually take place. The process to transform the phenotype into the genotype is called phenotype-genotype mapping and the opposite is called genotype-phenotype mapping. A place-holder is commonly called variable, a locus, a position or a gene. An object on such a place can be called value or an allele.

Inside EAs there are a set of components or operations that are considered basic components: fitness function, population, selection mechanisms, reproduction operators and initialization and finish condition. These terms are defined in the next lines:

*Fitness Function*: This function is also called evaluate function. It is the basis for the selection operations. It usually assigns an integer number to an individual. This number gives a measure of the quality of the individual; it tells how good that candidate solution is. Once we have the result for all the candidates, it is possible to get an order among all of them thanks to the fitness function returns an integer.

*Population:* A population is a set of individuals (genotypes). It forms the unit of evolution. Individuals are static objects not changing or adapting, it is the population that does. The diversity of a population is a measure of the number of different solutions present on it.

*Selection Mechanisms:* The selection mechanisms are used for two phases in the general scheme. In one hand it is valid to select among the individuals those that will be the parents to create the next generations. In the other hand it is also valid to select among all the individuals those that will be present in the next generation.

One property of the selection is that all of the individuals have a positive chance of being selected (this is a property that makes the evolutionary algorithms stochastic). Acting in this way make easier that if an individual get a local optimum can go out of it, otherwise it is possible to interpret it like a global optimum.

There are several different mechanism to achieve the selection of the individuals [23]:

- Tournament selection. Tournaments are playing between two solutions and the better solution is chosen and placed in the set of the selected individuals (mating pool). In the general implementation, the individuals for each tournament are selected randomly but each individual can participate only in two different tournaments, so at the end, we could have in the mating pool two copies (if the individual wins two times), one copy (if the individual wins one time) or zero copies (if the individual doesn't win). The number of tournaments done is the same as the number of individuals that we want to have as a parent for the next generation (if we are selecting parents) or the same as the number of individuals that we need to have in the next generation (if we are selecting the individuals for the next generation).
- Proportional selection. Copies are assigned to the solutions, the number of which is proportional to their fitness value. As example we can see the following table, if we have five individuals with their respective fitness value (first and second columns), we must complete the remainder columns, we have to fill the third and the fourth columns:

| Solution | Fi (Fitness) | pi = Fi/SUM(Fi) | Pi = SUM(pi) |
|---|---|---|---|
| **1** | 25 | 0.25 | 0.25 |
| **2** | 5 | 0.05 | 0.3 |
| **3** | 40 | 0.4 | 0.7 |
| **4** | 10 | 0.2 | 0.8 |
| **5** | 20 | 0.2 | 1 |

**Table 1. Proportional Selection**

Once we have the table complete, we can use the concept of roulette wheel selection [27] to select the individuals for the mating pool. As it is shown in

Figure 10, we create a random number between 0 and 1, and then we map this number in the roulette. The individual that is in the space that belongs to the random number will go to the mating pool and then we create another random number and repeat the process N times (N is the number of individuals that we need in the mating pool). Figure 10 shows the roulette wheel for the Table 1, e.g., if the random number were 0.21 the selected solution in this case will be the 1.

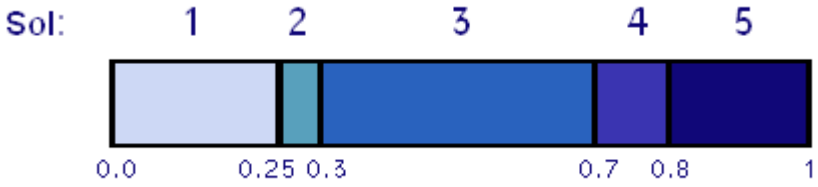


**Figure 10. Roulette Wheel Selection**

This method has several improvements like eliminate the variance or use a stochastic universal sampling [23]. The basic problem with the methods based in the roulette wheel selection is that the outcome is dependent only of the true fitness value instead of the relative fitness value of the population members. Tournament selection does not have this problem.

- Ranking selection: This method makes an order list of the solutions according their fitness values. Each individual has a weight that will be inversely proportional to their position. As an example the first individual will have the weight N and the last individual in the list will have the weight 1. Then it does the proportional selection operator with these weights.

*Reproduction operators:* The role of the reproduction operators is to create new individuals from the old ones. There are unary (mutation) or n-ary operators (recombination or reproduction):

- Mutation. One chromosome is selected and one or more of their genes will be changed. Each gene of the chromosome has a positive chance of being elected to be mutated. This operation is important to preserve diversity in a population because it creates individuals with new information. Translating into the search terminology, use the mutation is useful to explore new areas far from the actual region explored, acting in this way is easier to avoid local optimum. Figure 11 shows the mutation operator for a single point and for multiple points.

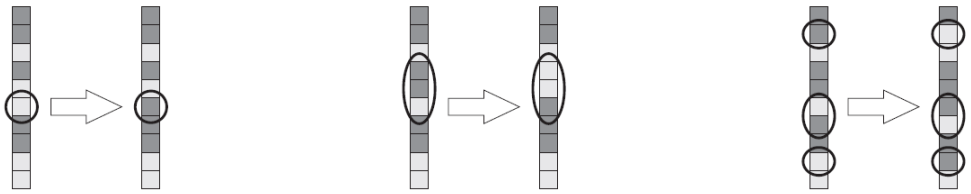

**Figure 11. Mutation operations of a simple and multiple genes [1]**

- Reproduction. It is the n-ary reproduction operator and is also called recombination. Usually it is a binary operator like in the nature, when there is no possibility of find species than need more than two individuals to practice the reproduction, but in the evolutionary algorithms it is possible to do the reproduction among more than two individual. This is also a stochastic operator because as in the mutation all the genes have a positive chance of being elected to do the recombination. The mechanism selects two or more random individuals, and then the operator merges the information from two genotypes in one or more offspring genotypes. The principle is combine features to get a new individual. In this case in contrast to mutation the operation creates new individuals but with the information of the old ones. Thus, in this case the new exploration is done in the same region but in a deep way, for example if we are near an optimal, with this operator we will get it. Figure 12 shows several situations for recombination depending on the number of points that defines the information that had to be changed between the different individuals.

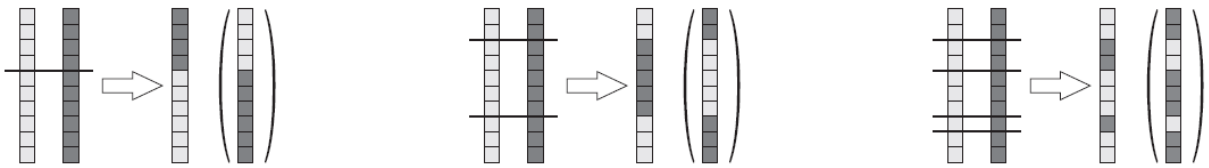

**Figure 12. Recombination operations [1]**

*Initialization and termination condition:* The first population is seeded by randomly generated individuals. It is possible the use of heuristics to guide the first populations, but seems not to be so effective because the population advances fast at the beginning towards a determinate fitness value, even when the initialization doesn't include heuristic, so use heuristic or not depend on the computational effort that it needs [28].

Regarding the termination condition, it is possible to fix an optimal known fitness level, but the problem here is that as EA are stochastic and hence it is possible that the decided fitness level could not be ever reached. To avoid this situation is needed to add other mechanisms for the termination condition like the CPU time elapsed, the number of fitness evaluations, the drop of the population diversity, etc. It is important to note that it might not be worth to allow very long runs: due to the any-time behavior on evolutionary algorithms, efforts spent after a certain time may not result in a better solution quality [28].

Once the main elements of the EA have been presented, a general scheme for the EA is shown (Figure 13). Note that EA is a wide area and probably there are a lot of different schemes, but all the EA follow in more or less detail this scheme.

```
Begin
  Init the first population randomly;
  Evaluate each individual;
  Repeat
    Select individuals for the reproduction;
    Reproduction (mutation, recombination or both);
    Evaluate new indviduals
    Select individuals for the new population;
  Until (finish condition=true)
End
```

**Figure 13. Scheme for EA**

Inside EA there are several kinds of algorithms. They have been classified according their properties and it is possible to distinguish among four categories: genetic algorithms (GA), genetic programming (GP), evolutionary programming (EP) and evolution strategies (ES). In the next paragraphs a brief description of each of these branches of the EA is given to have a global vision of EA.

*Genetic algorithms (GA):* Is a subset of the EA where the elements of the search space (the genotypes) are usually binary strings and rarely arrays of other kind of elements. The genotype is used in the reproduction operators while the fitness function is computed based in the phenotype, obtained thanks to the genotype-phenotype mapping.

This branch of the EA was created based on the ideas of Fraser [29] and developed by Holland [30]. The genetic algorithms are the main exponent of the evolutionary algorithms.

*Evolution strategy (ES):* Evolution Strategies (ES) was introduced by Rechenberg [31] with the following features:

- They usually use vectors of real numbers as solution candidates.
- Mutation and selection are the primary operators and recombination is less common.
- The selection is deterministic and only based on the fitness rankings, not on the actual fitness values. The simplest ES operates on a population of size two: the current point (parent) and the result of its mutation. Only if the mutant has a higher fitness than the parent, it becomes the parent of the next generation. Otherwise the mutant is disregarded. This is a (1+1)-ES. More generally, $\lambda$ mutants can be generated and compete with the parent, called $(1 + \lambda)$-ES. In a $(1, \lambda)$-ES the best mutant becomes the parent of the next generation while the current parent is always disregarded. There are some more combinations to get the next generation using more than one parent.
- In all other aspects, they perform exactly like basic evolutionary algorithms

*Evolutionary programming (EP):* It was first used by Fogel [32] in order to use simulated evolution as a learning process aiming to generate artificial intelligence. Fogel experimented with the evolution of finite state machines as predictors for data streams. Currently there exists no clear specification or algorithmic variant for evolutionary programming. It is becoming harder to distinguish from evolution strategies. Mutation and selection are the only operators used in evolutionary programming and recombination is usually not used.

*Genetic Programming (GP):* The words genetic programming (GP) [33] can have two meanings. The first of the meanings refers to all the EA that are capable of create programs, algorithms and similar constructions. Sometimes we have the inputs and their correspondent solutions for a problem and we want to obtain the intermediate program that for the provided inputs obtains the provided solutions.

The second meaning refers to all the EA that has as a genotype tree data types. These ideas came from Koza [33] and made GP more popular because it is possible to codify a great variety of problems. A tree can represent a set of rules, a mathematic expression, a decision tree, etc

In GP the reproduction operations are quite different because in this case the genotypes are trees. Figure 14 and Figure 15 show examples of recombination and mutation applied to the tree genotypes.

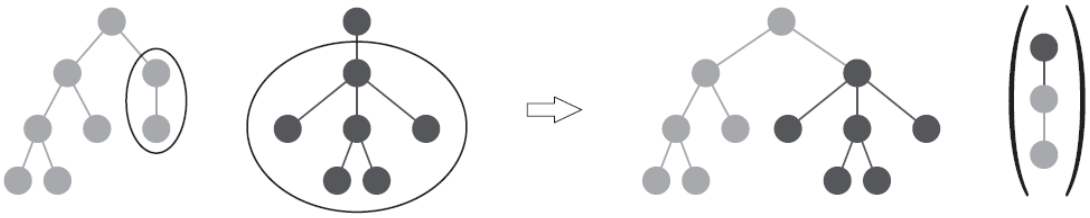

**Figure 14. GP Recombination**

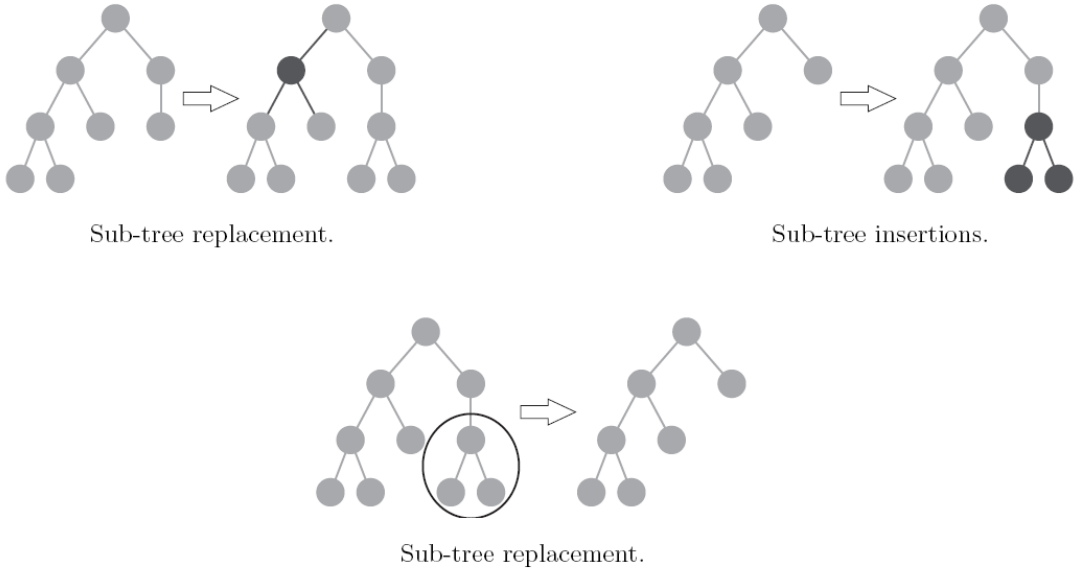


**Figure 15. GP Mutation**

Inside GP there are several variations of the original approximation. Among them it is grammatical evolution (GE) that uses grammars as input. We have created a grammar that covers the DMM design space. Thus, we have used the GE algorithms in the automatic exploration of the DMM design space. For this reason GE is described in the next section.

### 2.2.2.3 Grammatical Evolution

The main feature of the GP is the use of trees as genotype and one characteristic is that it is possible to create programs, but there is one problem: it is possible to create invalid individuals. This problem is important because as more correct individuals we have, sooner we get the final solution. Figure 16 present an example of this problem: the division [1].

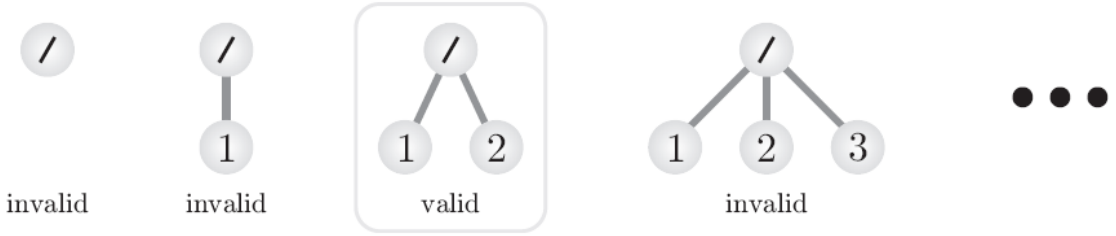


**Figure 16. Division Problem in GP**

The division is only valid between two numbers, other representation is a bad representation and not valid. If this is permitted, most of the individuals created will be invalids. From this point, some researches saw that the most effective way to tackle this problem was using special genotype-genotype mappings. To avoid the creation by the GP algorithms of invalid individuals they thought in something that express structural and semantic restrictions of a search space, they were thinking in the formal grammars. So a new kind of algorithm that combines principles from molecular biology (they belong to genetic programming) to the representational power of formal grammars appeared. These algorithms also have a rich modularity that gives a unique flexibility, making it possible to use alternative search strategies, whether evolutionary, deterministic or some other approach, and to radically change its behavior by merely changing the grammar supplied. Its genotype-phenotype mapping also means that instead of operating exclusively on solution trees, as in standard genetic programming, they allow search operators to be performed on the genotype (e.g., integer or binary chromosomes), in addition to partially derived phenotypes, and the fully formed phenotypic derivation trees themselves. From the first ideas in this field [33], several different implementation have arisen, one of them is the most interesting for this research work, grammatical evolution (GE)

GE, developed by Ryan [34] and extended in some researches and used in diverse fields [35, 36], creates expressions in a given language by iteratively applying the rules of a grammar specified in the Backus-Naur form.

As illustrated in Figure 17, a Grammatical Evolution system consists of three components: the problem definition (including the means of evaluating a solution candidate), the grammar that defines the possible shapes of the individuals, and the search algorithm that creates the individuals [37]

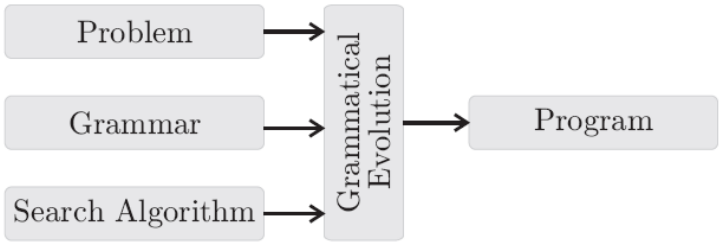


**Figure 17. GE System**

It is important to remember that a grammar specified in the Backus-Naur form can be represented by the tuple {N, T, P, S} where N is the set of non-terminals, T the set of terminals, P a set of production rules that maps the elements of N to T, and S is a start symbol which is a member of N. When there are a number of productions that can be applied to one element of N the choice is delimited with the "|" symbol.

A GE's individual uses a variable-length encoding scheme where each gene holds an integer value that will be mapped to previously labeled production rules of a given BNF by the decoding process. The genotype is used to map the start symbol as defined in the Grammar onto terminals by reading genes to generate a corresponding integer value.

```
N = {<GlobalHeap>, <HeapList>, <Heap>, <Header>,
     <AllSel>, <FreeSel>, <Tail>}
T = {SingletonHeap, (, ), FIFOFirstFit,
     FIFOBestFit, Header1, Header2,
     AllSel1, AllSel2, FreeSel1, FreeSel2, HeapOS}
S = <GlobalHeap>
P = I   <GlobalHeap> ::= SingletonHeap(<HeapList>)   (0)
    II  <HeapList>   ::= HeapList(<Heap>,<AllSel>,
                                  <FreeSel>,<Tail>)  (0)
    III <Heap>       ::= FIFOFirstFit(<Header>)      (0)
                       |FIFOBestFit(<Header>)        (1)
    IV  <Header>     ::= Header1                     (0)
                       |Header2                      (1)
                       |Header3                      (2)
    V   <AllSel>     ::= AllSel1                     (0)
                       |AllSel2                      (1)
    VI  <FreeSel>    ::= FreeSel1                    (0)
                       |FreeSel2                     (1)
    VII <Tail>       ::= HeapList                    (0)
                       |HeapOS                       (1)
```

**Figure 18. Example of grammar**

To see how the GE genotype-phenotype mapping works, in Figure 18 is shown a grammar that defines an abstract language for DMMs implementation based in the library presented before. The grammar is the set of the N, T, P and S. N is the set of non terminals (GlobalHeap, HeapList, Heap, Header, AllSel, FreeSel and Tail), T is the set of terminals (SingletonHeap, (, ), FIFOFirstFit, FIFOBestFit, Header1, Header2, AllSel1, AllSel2, FreeSel1, FreeSel2 and HeapOS) , S is the start symbol (GlobalHeap), and P is the set of seven productions that indicates the rules that must be followed to create a correct composition.

Now consider the follow genome:

| 204 | 143 | 55 | 224 | 15 | 7 | 75 | 191 | 233 | 1 |
|---|---|---|---|---|---|---|---|---|---|

**Figure 19. Example of individual**

There are 10 genes with values ranging from 0 to 255 (8-bit number). Since an 8-bit integer is far more than the number of production rules, the modulus operation is needed to decode the genes properly.

The decoding process starts in the first rule (group I):

```
<GlobalHeap> ::= SingletonHeap(<HeapList>
```

In each step the algorithm reads a gene from the individual (in this case 204) and from this gene decides what is the production selected for compose the final phenotype. To decide this it is necessary to do the module operation between the information extracted from the gene and the number of productions that there are in the actual rule. In this case the operation would be:

```
204 mod 1 = 0
```

So the production selected (it was trivial because in this case there was only one production) is:

```
SingletonHeap(<HeapList>)
```

The next non-terminal that there is in the actual phenotype is HeapList, so the algorithm must go in the rule for HeapList:

```
<HeapList> ::= HeapList (<Heap>, <AllSel>, <FreeSel>, <Tail>)
```

Now the algorithm takes the next gene and decodifies it:

```
143 mod 1 = 0
```

In this case as in the previous one, like there is only one production, that is the selected one:

```
HeapList (<Heap>, <AllSel>, <FreeSel>, <Tail>)
```

And the phenotype at this step is:

```
SingletonHeap (HeapList (<Heap>, <AllSel>, <FreeSel>, <Tail>))
```

At this point there are four possible non-terminals, and the algorithm follows an order, from the left to the right, so the next rule is:

```
<Heap> ::= FIFOFirstFit(<Header>) | FIFOBestFit(<Header>)
```

The decodify process in this step is:

```
55 mod 2 = 1
```

The selected is the second production, so at this point the phenotype is:

```
SingletonHeap (HeapList (FIFOBestFit(<Header>), <AllSel>, <FreeSel>, <Tail>))
```

The next steps will sequentially produce the following expressions:

```
224 mod 3 = 2
SingletonHeap(HeapList(FIFOBestFit(Header3), <AllSel>, <FreeSel>, <Tail>))
15 mod 2 = 1
SingletonHeap(HeapList(FIFOBestFit(Header3), AllSel2, <FreeSel>, <Tail>))
7 mod 2 = 1
SingletonHeap(HeapList(FIFOBestFit(Header3), AllSel2, FreeSel2, <Tail>))
75 mod 1 = 0
SingletonHeap(HeapList(FIFOBestFit(Header3), AllSel2, FreeSel2, HeapOS))
```

At this point there are no more non-terminals so the decodify process has ended and the final phenotype is SingletonHeap(HeapList(FIFOBestFit(Header3), AllSel2, FreeSel2, HeapOS)).

In the case of GE as it has been shown, the representation (genotype) is not a tree itself although the phenotype can be represented with it. The advantage here is that GE algorithms can directly use all standard genetic algorithm operators that are less complex. Furthermore, because of the simplicity of the linear representation, computer implementations of GE are relatively easy.

In the presented example the decoding process finishes without translating all genes. This occurs because in grammatical evolution the genetic operations do not know about the semantics of a genome until it is decoded, so the decoding process frequently ends up with a complete expression (final) without traversing the entire genome. But a serious problem arises if the decoding process has not ended when a complete individual has been read. If there are not

enough genes to generate a complete phenotype, three paths have been described to afford this situation

- Mark the genotype as invalid and give it a reasonable bad fitness.
- Expand the remaining non-terminals using default rules (i.e., we could say that the default value for Heap is FIFOFirstFit(<Header>), for Header is Header1, for AllSel is AllSel1, for FreeSel is FreeSel1 and for tail is HeapOS).
- or wrap around and restart taking numbers from the beginning of the genotype. If this option is chosen it is possible that even after wrapping, the mapping process would be incomplete and would carry on indefinitely unless terminated. This occurs because a non-terminal is being mapped recursively by a production rule. Such an individual is dubbed invalid as it will never undergo a complete mapping to a set of terminals. For this reason an upper limit on the number of wrapping events that can occur is imposed. This is the option selected for this research work.

The next step is to merge together all the contents seen until now. In this chapter has been shown an introduction to the basic concepts of optimization, also some approaches to MOOP, after that an introduction to the EA has been given, and finally the GE algorithms have been explained. Now with the basis of GE in the next chapter is presented the methodology that use GE to obtain optimal DMMs.

# 3. DMM OPTIMIZATION FLOW

The central point of this work is to take all the previous ideas, put it together and create a flow to create optimal DMMs. The final optimization framework proposed has three different phases to perform the automatic exploration of DMMs using Grammatical Evolution (GE). In each of these phases a different tool is used, but all the process is done without the need of human interaction. The designer is only needed when the flow start, to provide some hardware parameters, and it is possible to interact in other some points in the flow to help the search process, but this interaction is not necessary. Figure 20 shows the different phases required to perform the overall DMMs optimization. The optimization flow starts with the profiling phase. In this phase a profiling report that includes important information about the execution of the application is created from the original application. In the next step, the grammar filter phase, the grammar tool will create a grammar taking in account the data extracted from the profiling and the data requested to the designer. The grammar created is a specialized grammar for the application and final embedded system under study. Consequently, such phase also reduces the search space. Finally, in the third phase an exploration of the design space of DMMs implementation is performed using GE. Next, we describe the three phases of our proposed optimization flow.

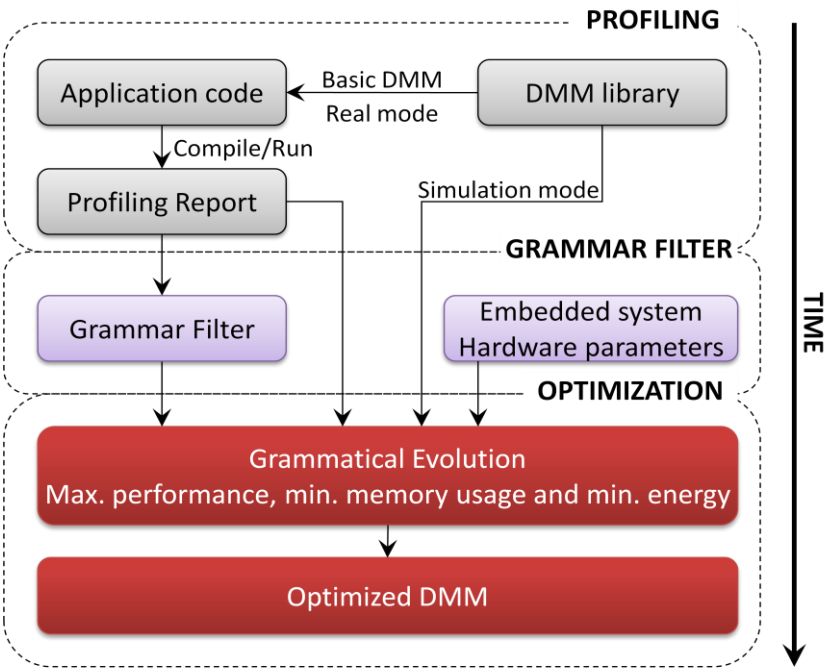


**Figure 20. DMM Optimization Flow**

# 3.1 Profiling of the application

In the first phase, the application under study is running using a basic DM manager implemented with the DMM library. To this end, we must only include the DMM library in the source code of the application (one line of code per variable to profile). The DMM library includes a layer that makes easier the process of collect information about the application. When the application finishes its execution, a file is created including important information about the behavior of the application. Among the information is included all the memory operations, like objects created, deleted read or written. For each of the operation it stores the kind of operation (access, allocation or deallocation), the accessed object size and the memory address as proposed in [9]. As a result, this first phase takes between 2-3 hours in our methodology for real-life applications, thanks to our tools with very limited user interaction.

# 3.2 DMM grammar filter

The final result of this second phase, the grammar filter, is as the name indicates a grammar. It is necessary to come back and remember that to use GE algorithms it is compulsory to use a grammar. In the chapter 2.2.2 Evolutionary Algorithms, an abstract grammar to construct DMMs was presented as an example. We have created a tool called grammar filter that will create automatically a grammar that cover the DMMs design space, specific for the parameters that have been received. In Figure 22 we can see a hypothetical possible grammar that could be formed by the grammar tool.

```
<GlobalHeap> ::= <SingletonHeap>
<SingletonHeap> ::= SingletonHeap(<CoalesceHeap>)|SingletonHeap(<HeapList>)

<HeapList>  := HeapList(<Heap>,<Selectors>,<Tail_Size>)
             | HeapList(<Heap>,<True_Selectors>,<Tail_True>)

<Heap> ::= FIFODLFirstFitHeap>
        |<FIFODLBestFitHeap>
        |<FIFOSLFirstFitHeap>
        [… ]
        |<LIFOSLBestFitHeap>

<FIFODLFirstFitHeap> ::= FIFODLFirstFitHeap(<SizeHeader>)
<FIFODLFixedListHeap> ::= FIFODLFixedListHeap(<SizeHeader>)
[… ]
<LIFOSLBestFitHeap> ::= LIFOSLBestFitHeap(<SizeHeader>)

<Tail_Size> ::= <HeapList>|<CoalesceHeap>
<Tail_True> ::= <FixedHeap>|<CoalesceHeap>

<CoalesceHeap> ::= CoalesceHeap(<HeapList_Coalesce>,<MinSize>,<MaxSize>)
<MinSize> ::= #values#
<MaxSize> ::= #values#
<HeapList_Coalesce> ::= HeapList(<Heap_Coalesce>,<Selectors>,<Tail_Coalesce_Size>)
                      | HeapList(<Heap_Coalesce>,<True_Selectors>,<Tail_Coalesce_True>)

<Heap_Coalesce> ::= <FIFODLFirstFitHeap_Coalesce>
                 |<FIFODLBestFitHeap_Coalesce>
                 [… ] |<LIFOSLBestFitHeap_Coalesce>

<FIFODLFirstFitHeap_Coalesce> ::= FIFODLFirstFitHeap(<LeaHeader>)
[… ]
<LIFOSLBestFitHeap_Coalesce> ::= LIFOSLBestFitHeap(<LeaHeader>)

<Tail_Coalesce_Size> ::= <HeapList_Coalesce>
<Tail_Coalesce_True> ::= CoalesceableHeap(<SlopHeap>)

<True_Selectors> ::= <True_Selector>, <True_Selector>
<True_Selector> ::=  TrueSelector

<Selectors> ::= <Lt_Selectors>
               |<Lte_Selectors>
               [… ] |<Size_Selectors>

<Size_Selectors> ::= #values#
<Lt_Selectors> ::= #values#
[… ]
<Lte_Selectors> ::= #values#

<EmptyHeader> EmptyHeader
<SizeHeader> ::= SizeHeader
<LeaHeader> ::= LeaHeader

<SlopHeap> ::= SlopHeap(<FixedHeap_Coalesce>, <LeaHeader>)
<FixedHeap_Coalesce> ::= FixedHeap(<SbrkHeap>,<MemorySizeInKB>,<EmptyHeader>)
<FixedHeap> ::= FixedHeap(<SbrkHeap>,<MemorySizeInKB>,<SizeHeader>)
<SbrkHeap> ::= SbrkHeap(<EmptyHeader>)
<MemorySizeInKB> ::= #MemSize#
```

**Figure 21. Excerpt of DMM grammar**

In particular, the shown grammar (Figure 21) is one created by the grammar filter, but some options are not included due to simplify the understanding. Nonetheless, this grammar is complete enough to implement many well-known DMMs and to explore custom DMM implementations for the two real-life case studies used in this work. In the next paragraphs a brief introduction to the components of the grammar is given.

The CoalesceHeap heap indicates that for all the heaps included on it, it is allow the splitting and coalescing mechanisms and it determines the maximum and minimum sizes of the blocks after these mechanisms. When this heap is included in a DMM, it must also include the CoalesceableHeap and the SlopHeap that are layers that help to maintain the memory when the splitting and coalescing are used. There are some rules in the grammar that seem to be quite similar to another (e.g. FIFODLFirstFitHeap and FIFODLFirstFitHeap_Coalesce). This is because some rules must be redefined if there is a coalesceHeap, i.e. is needed to include the LeaHeader (that let to manage the blocks) in some rules.

The headers are the extra information added to each block to maintain some information. Three headers are included: EmptyHeader which represents just the object, SizeHeader that maintains object size in a header just preceding the object, and LeaHeader that does the same but also records whether each object is free in the header of the next object in order to facilitate coalescing.

The HeapList heap represents the structure defined in the chapter 2.1 Dynamic Memory Managers. It includes a heap, a selector and a tail. Concerning the heaps they are formed as a combination of a policy for the insertion and extraction (FIFO or LIFO), a kind of list (Single linked or double linked) and a policy for decide the block inserted or extracted (BestFit, FirstFit, FixedFit, or WorstFit). Thus, a possible heap could be for example, FIFODLFistFitHeap, which represents a heap that have a double linked list for the free blocks and when a requesting of a block is received, it returns the first block that fits starting the search from the first block inserted. The selectors decide if a block could be inserted or extracted from that heap. There are some different selectors (less than, less or equal than, great than, great or equal than and equal than). A possible selection decision could be Gt_Selection (60) (this heap only accepts blocks with a size bigger or equal to 60 bytes). The final part of the HeapList, the tail can be other HeapList (to add other heap with different properties) or if the DMM is finished, it will be the interface with the OS (SbrkHeap, built using sbrk() for UNIX systems and a sbrk() emulation for Windows).

There is a particularity in the HeapList. The particularity is that if the HeapList is part of a CoalesceHeap, then the mechanisms of coalescing and splitting can be used, but otherwise these mechanisms cannot be used. And it is possible to form a DMM where the first heaps are

HeapList and after that a CoalesceHeap is included and for all the heaps that are beyond this point the coalescing and splitting mechanisms are allow, but not for the rest.

The grammar tool, based in the information extracted from the profiling, the embedded hardware parameters introduced and optionally some decisions of the designer create the final grammar used in the next phase. There are some rules that are not as general as the productions presented before, these rules must be filled by the grammar tool and they will be specific for each application. The grammar tool must decide which are the proposed sizes for the minimum and the maximum size for coalescing. It must decide also what sizes are possible for the lists of free blocks. It takes the information contained in the profiling report and after some computations, decides the final information added to the grammar. The places to fill are marked in the grammar presented (Figure 22) as #values#. Other inputs to the grammar tool are the embedded hardware parameters. They are used to fill the #memSize# sentence. Finally if the designer because of his experience or by any other reason thinks that is better not to use some of the options, e.g. he thinks that for this application is not necessary to explore the DMMs that include the worst fit policy, then he can deactivate this option reducing in this way the search space. But this step is optional; in fact the experiments have been done automatically, without the designer interaction obtaining promising results.

In the final experiments, without the designer interaction, the duration of this phase is no more than 1-2 minutes.

## 3.3 Optimization

The last phase is the optimization process. As Figure 20 depicts, this phase consists of a GE algorithm that takes as input:

- the grammar generated in the previous phase,
- the hardware parameters (e.g., memory size and power consumption model for the embedded memory [38]) of the target embedded system,
- and the profiling report of the application.

It also uses the DMM library, extended to simulate the behavior of every DMM generated by the grammar when it is used in the application without the need of recompile the application each time that the DMM changes. The next figure (Figure 23) shows an illustrative example on how our methodology performs. The GE algorithm creates a population of individuals, for each

individual the algorithm performs the genotype-phenotype mapping (gpm), just like it has been explained before (chapter 2.2.2 Evolutionary Algorithms), the algorithm takes each of the chromosomes and decodes it, gene by gene getting the corresponding rules from the grammar to create the final phenotype for each genotype. Like the grammar presented before is used to describe a DMM, a phenotype will represent a DMM (In the Figure 23 $DMM_i$) that will be passed to the DMM library.

The DMM library has been extended with a simulation mode option and if this option is selected the library will emulate the behavior of the application. To do that the application read each line from the profiling report and emulate how the real DMM would work. Such emulation does not de/allocate memory from the computer like the real application, but maintains useful information about how the structure of the selected DMM evolves in time. Such methodology is much faster than previous approaches proposed in the literature [10, 11], and allows the system designer to use automatic exploration algorithms instead of compiling and running the application for every new DMM. After all the profiling has been simulated, the DMM library returns back the fitness of the current DMM to the GE algorithm.

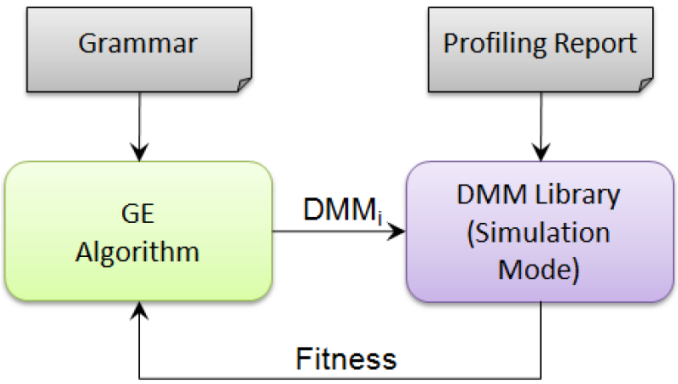


**Figure 23. DMM Generation and Evaluation Process**

The fitness of a given DMM is obtained as a weighted sum of the performance, memory usage and energy consumed by the DMM for the target embedded system and application under study. The memory usage is directly calculated by the simulator while is emulating the behavior of the DMM, each storage in memory done by the DMM (could be the memory necessary by the DMM to work, like lists maintenance or could be the memory used to satisfy the requests done from the profiling) is computed. Regarding the performance, we assume that is inversely proportional to the execution time of the application using the DMM being evaluated ($t_{ex}$):

$$\text{Performance}\,(DMM_i) \propto {}^{1}/_{t_{ex}(DMM_i)}$$

To measure the execution time of the application under study (and using the DMM proposed by the GE algorithm), the DMM simulator calculates the computational complexity or time complexity [39]. Considering the energy consumption, we have assumed that is proportional to the number of memory accesses. We know that there are more facts that cause high power consumption, but for this research we think that is enough attending only at the memory accesses. In several research works, cache misses have been computed in order to estimate the energy consumed by applications [6, 9, 18, 40]. However, cache misses are costly to compute, since it requires a complex analytical model or a cache-simulator. In this work, we are interested in evaluating a DMM as fast as possible, while being able to explore global trade-offs between different DM managers, so we mainly focus in (accurate enough) high-level exploration, rather than evaluating cycle by cycle the execution of the application in the internal buses of the embedded system. Furthermore, as it is showed in [40], the most important factor in an energy model is the execution time and the number of memory accesses, which we model accurately.

To measure all the previous metrics, every portion of the code in the simulator that emulates the behavior of a DMM is accompanied by its corresponding added execution time, memory accesses and memory used. The following code snippet (Figure 24) shows an illustrative example of how this task is performed:

```
inline void malloc(size_t sz) {
        exTime += 2; memAcc += 2;
        object* ptr = head.next;
        if(ptr!=&tail) {
                exTime += 2; memAcc += 4;
                head.next = ptr->next;
                if(head.next==&tail) {
                        exTime++; memAcc += 2; memUsed -= ptr->size();
                        tail.next = &head;
                }
                exTime++;
                return (void*)ptr;
        }
        exTime++;
        return 0;
}
```

**Figure 24. Metrics updating example**

The first sentence computes the execution time of the pointer assignment (ptr) and the evaluation of the if condition. The second one takes into account two memory accesses: one for the head.next sentence (i.e., access operator) and one because of the &tail sentence. This process is repeated until the end of the function, updating the execution time, memory accesses and memory used when needed. In the example presented, there is a sentence where the

memory used is reduced in ptr->size(): this is correct because the DMM does not need to manage this portion of memory, unless it is freed.

When the optimization process ends, the GE algorithm returns the best DMM found, with minimal weighted sum of memory accesses, memory used and energy consumed. This phase takes no more than few hours with no user interaction. It mainly depends on the size of the profiling report. In the performed tests, the size of the profiling reports varying from 2.4 to 3.1 GB. Note that in previous approaches, this phase typically takes days, and requires that the application does not demand user interaction [10]. In any case, this methodology requires less time than state-of-the-art solutions to the same problem [10] because this work is done with a profiling report, instead of simulating multiple times the complete original application. Furthermore, the original application is not compiled every time a new DMM must be evaluated, which makes this framework even more stable and results more easily comparable overall.

## 3.4 Experimental Results

### 3.4.1 Case studies

The proposed methodology has been applied to two case studies that represent different modern multimedia application domains. The first case study is a 3D Physics Engine (Physics3D) for elastic and deformable bodies [41], which displays the 3D interaction of non-rigid bodies. The second benchmark is VDrift [42], which is a driving simulation game. The game includes as main features: 19 different tracks, 28 types of cars, artificial intelligent players and a networked multi-player mode.

To know the quality of the solutions obtained, a comparison is done among the new proposed methodology, two well-known DMMs, and a previous approach to obtain optimized DMMs using heuristics [3].

The Lea Allocator (for further information see Chapter 2.1) is one of the most known, used and complex allocators. It is among the fastest while also being among the most space-conserving, portable and tunable. These reasons make Lea Allocator a good general-purpose allocator for malloc-intensive programs.

The Kingsley Allocator (for further information see Chapter 2.1) is a well-known memory allocator. It is one of the fastest general-purpose allocators although it is among the worst in terms of fragmentation.

The cDMMs manual optimization using heuristic has been called classic approach for simplicity. This approach, instead of being a general purpose DMM like the two previous DMMs, is a methodology to obtain a custom DMM. The methodology is proposed in [10] and it is the basis for this research work. The process to create new specific custom DMMs has several steps. First of all it is necessary to collect information from the application under study. With this information some decisions must be taken to reduce the DMMs design search space and to follow an order to explore it. The authors of [10] use several heuristics to make the decisions but this phase is really time-consuming. After this step several DMMs are created, and then it is necessary to evaluate it. The evaluation is done running the original application including the DMM generated. In this process it is necessary to compile the application and execute it. Finally the DMM that obtains the best results in terms of performance, energy consumption and memory usage is selected as the optimal DMM. This final DMM is the one used in the comparison.

Therefore the comparison will be done among: one of the fastest DMMs (Kingsley), one of the most space-conserving DMMs (Lea), one application specific DMM and the DMM generated by the optimization flow proposed in this research work. For the GE algorithms the parameters are given in the Table 2

| Parameter | Value |
|---|---|
| **Population Size** | 60 |
| **Number of Generations** | 100 |
| **Probability of Crossover** | 0.80 |
| **Probability of Mutation** | 0.02 |

**Table 2. GE algorithm parameters**

### 3.4.2 Method applied to Physics3D

To create the final custom DMM, the proposed methodology flow has been followed. The first step is to profile the behavior of the application using a basic DMM implementation. Then, the Grammar Filter tool is executed and a reduced grammar file is obtained. Finally with this grammar, the profiling report and the hardware embedded parameters the GE algorithm is executed obtaining an optimal DMM. As this process is stochastic due to the non-deterministic nature of the GE algorithms, the explained flow has been repeated ten times and the mean of each of the metrics has been selected as a representative value of the final result. The metrics in study are the performance, the energy consumption and the memory usage. In the Figure 25 is plotted the comparison among the above mentioned approaches for each of the metrics normalized to the Kingsley results.

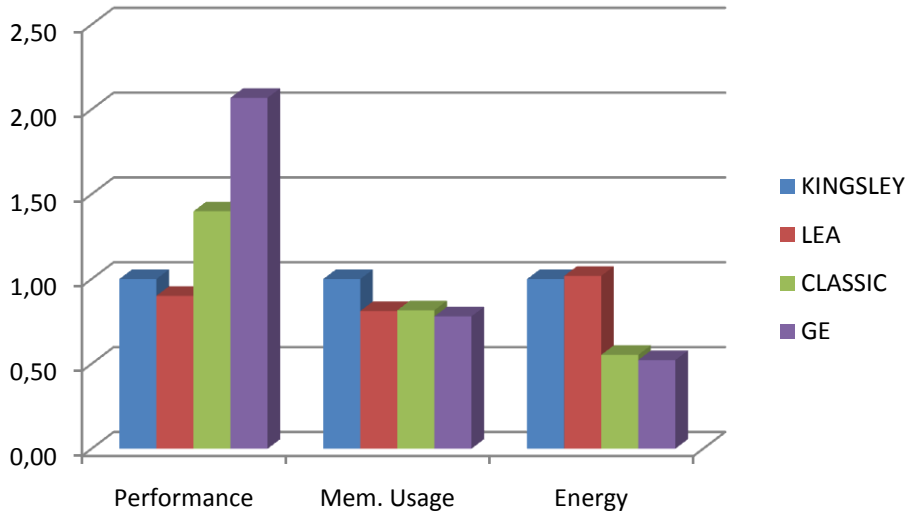


**Figure 25. Physics3D – Comparison among Kingsley, Lea, Classic and GE approaches**

Some conclusions can be extracted from the Figure 25. Regarding the performance, the DMMs obtained by the optimization flow presented in this work obtain the best result (more than 30% better than the classic approach, 52 % better than Kingsley and 56% better than Lea). Concerning the memory usage, the worst result is obtained by the Kingsley allocator due to the structure of list that it maintains even when the lists are not used. The approach that uses less memory is the approach proposed in this work. The reason is that it only creates the lists that will be needed and it uses the coalescing and splitting mechanism in an effective way. It uses a 4% less memory than the classic approach and Lea Allocator and it uses a 20% less memory

than the Kingsley approach. Finally looking at the energy consumption, the DMMs obtained in this research work obtain the best result. The less energy consumption is because as is an application specific DMM, it creates specific structures, and less memory accesses are needed, then the energy consumption decreases. In this case the GE approach obtains almost the same result than the classic approach (it only outperforms in 0.5%) but Lea and Kingsley consume the double of energy.

In addition to the previous study, it is necessary to add a times study. In this case the comparison is done between the DMM design optimization flow presented in this work and the classic approach that are the two methodologies. In the Figure 26 are shown the times needed by each of the flows to create an optimal DMM, normalized to the classic one. The results show that due to the fact that the new flow doesn't need the interaction with the designer and due to the simulation mode included in the DMMlibrary the time needed to create an optimal DMM has been reduced in a 95% obtaining a speed-up of 25.

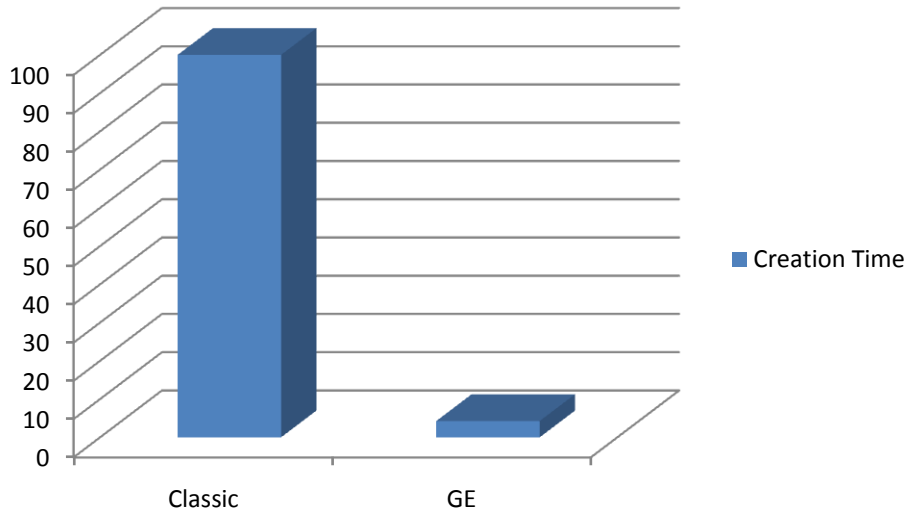


**Figure 26. Physics3D – Creation times study**

These results shown that in mean the DMMs obtained by the proposed DMM optimization flow outperforms the results obtained by other well-known general purpose allocators and even outperforms the DMM generated specifically for this application by a previous methodology. In addition the time needed to create a better DMM is 25 times less than for the classic approach and without the designer interaction.

### 3.4.3 Method applied to VDrift

As for physics3D, the proposed methodology is followed to create the final DMMs to make the comparisons. First of all the profiling report is created and it contains the relevant information from the VDrift application. After this, the grammar tool creates a reduced grammar that represents DMMs. And finally using the GE algorithm, an optimized DMM for VDrift has been created. As in the previous application, a set of ten executions of the algorithm have been done to obtain the mean of all the results, because of the stochastic feature that characterize the GE algorithms one simple execution is not representative. In the Figure 27 are represented the results obtained by each of the DMM implementations (Kingsley Allocator, Lea Allocator, Classic approach and GE approach) for the energy consumption, the memory usage and for the performance normalized to Kingsley Allocator.

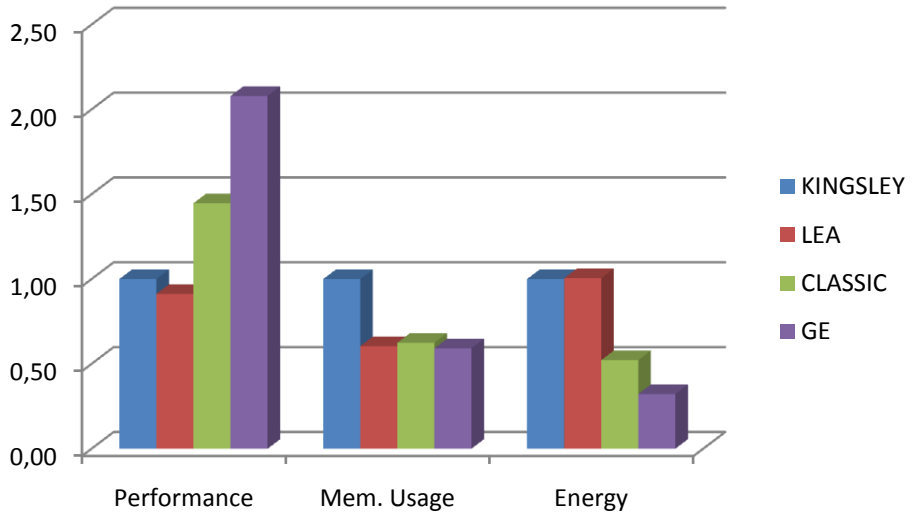


**Figure 27. VDrift – Comparison among Kingsley, Lea, Classic and GE approaches**

Processing the information from the Figure 27, it is possible to see that again the DMMs obtained by the optimization flow presented in this research work outperforms the result of the other approaches. Concerning the performance, the GE approach obtain a significant improvement in the performance compared to the general-purpose DMMs (For the Lea and Kingsley approaches it increases the performance in more than 50%), and compared to the classic approach it also obtains an improvement of 25% in this case. Looking at the memory usage, the worst approach is the Kingsley one. In this case the result is even worse than in the previous application because in this case the VDrift application has a small set of different sizes,

so the lists that are not used in the Kingsley Allocator are more than in Physics3D application. The other three approaches have a near-optimal use of the memory, they apply the coalescing and splitting when needed, avoiding in this way the extra memory wasted by the unused space in the blocks. But due to the heaps organization the GE approach obtain a little improvement (1% respect the Lea approach and 3% respect the classic approach). Finally attending to the energy consumption, Lea and Kingsley approach are quite similar in their consumption, and the GE approach is the less energy consuming implementation. It obtains a reduction of the energy consumed respect to the classic approach of 40% and respect to Lea and Kingsley of 70%.

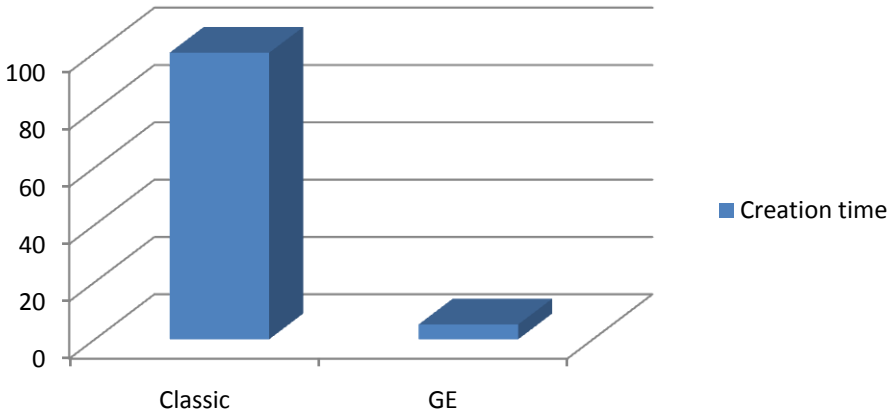


**Figure 28. VDrift – Creation times study**

The next relevant information is the time needed to create the final DMM in the cases that the DMMs must be created. In this case the comparison is done between the DMM design optimization flow presented in this work and the classic approach. In the Figure 28 are shown the times needed by each of the flows to create an optimal DMM, normalized to the classic one. The results as expected due to the fact that the new flow doesn't need the interaction with the designer and due to the simulation mode included in the DMMlibrary, the time needed to create an optimal DMM has been reduced in a 94% obtaining a speed-up of 19. In this case the less speed-up obtained compared with the speed-up obtained in the Physics3D application is due that as the VDrift has only a reduced set of different block sizes used during the execution, the complexity of the decisions to reduce the search space is less, and therefore it takes less time.

The conclusions extracted from these results are the same that the ones extracted from the Physics3D application. The optimized DMMs design flow explained along the current chapter obtains in a short period of time (compared to similar approaches) and without the need of the designer interaction a DMM that improves the results obtained by other approaches.

## 3.5 Conclusions

In this chapter we have presented our DMM optimization flow that has three different phases: the recollect of the information of a typical profile of the applications in execution, the creation of a grammar that redefines the DMM search space and finally the optimization phase that using GE algorithms obtain an optimized DMM to the system under study.

We have applied our methodologies to two case studies that represent different modern multimedia application domains. To compare our results, we have executed the applications with two well-known general purpose DMMs, one among the fastest and the other among the most space-preserving. In addition a previous methodology that uses heuristics to find optimal DMMs has been followed. The DMM obtained by this methodology has also been compared to the DMMs obtained by our DMM design flow.

As a result, our flow gets a DMM that obtain a better performance (always more than 25% in contrast to any of the other DMMs), a reduced memory usage (at least 1% less than any of the other DMMs) and a reduced energy consumption (at least 0.5% less than any of the other DMMs). But the main highlight of our methodologies is that these results are obtained with a speed-up in the time needed to create the DMM of at least 19. And the complete process has been done automatically, i.e. without the designer interaction.

# 4. INCLUDING PARALLELIZATION

## 4.1 Introduction

The process defined in the chapter 3 is automatic and faster than manual optimizations, but it demands intensive computation mainly in the simulation to obtain the fitness value, resulting in a very high time consuming process. In these cases the parallel processing approach can be very useful because it allow not only to explore more solutions spending the same time, but also to implement new algorithms.

Attending to the GE algorithms in this work, is proposed a parallel optimization scheme to the previous defined methodology (Chapter 3) to automatically generate optimal DMM implementations, thus improving the state-of-the-art exploration approaches.

The basic idea behind most parallel programs is to divide a task into chunks and to solve the chunks simultaneously using multiple processors. This divide-and-conquer approach can be applied to EAs in many different ways.

Existing parallel implementations of evolutionary algorithms can be classified into three main types [43]:

- Global single-population master-worker algorithms. In a master-worker EA there is a single population (just as in a simple EA), but the evaluation of fitness is

distributed among several processors. In this type of parallel algorithms, selection and crossover consider the entire population.

- Massively parallel algorithms, also called fine-grained algorithms consist of one spatially-structured population. Selection and recombination are restricted to a small neighborhood, but neighborhoods overlap permitting some interaction among all the individuals. The ideal case is to have only one individual for every processing element available.
- Distributed algorithms. Also called Multiple-population or coarse-grained parallel algorithms are more sophisticated, as they consist on several subpopulations which exchange individuals occasionally. This exchange of individuals is called migration and. Distributed algorithms are very popular, but also are the class of parallel GAs which is most difficult to understand, because the effects of migration are not fully understood. Distributed algorithms introduce fundamental changes in the operation of the GA and have a different behavior than simple EAs.

It is important to emphasize that while the master-worker parallelization method does not affect the behavior of the algorithm, the last two methods change the way the EA works. For example, in master-worker parallel EAs, selection takes into account all the population, but in the other two parallel EAs, selection only considers a subset of individuals. Also, in the master-worker any two individuals in the population can recombinate, but in the other methods recombination is restricted to a subset of individuals.

For the objective of this research work, the kind of algorithm that fit the best is the master-worker approach. The reason is that is easy to implement and take in account the entire population to make the selection and reproduction operations that will be better to obtain more heterogeneous individuals.

## 4.2. Parallel Implementation

In this section we describe how the GE algorithm proposed in the previous chapter works in a parallel environment to solve the exploration of DMMs in embedded applications. The search process can be significantly improved by using several threads to perform the simulations instead of running each one in the same processor. In this research work is proposed

a master-worker parallel GE (pGE) algorithm where each worker runs a different set of simulations (or fitness evaluations). The master processing element maintain the population and make the selection and reproduction operations while the evaluation of each individual is sent to another processing elements (workers) and once they have simulated the DMM, they will return the fitness value to the master (Figure 29).

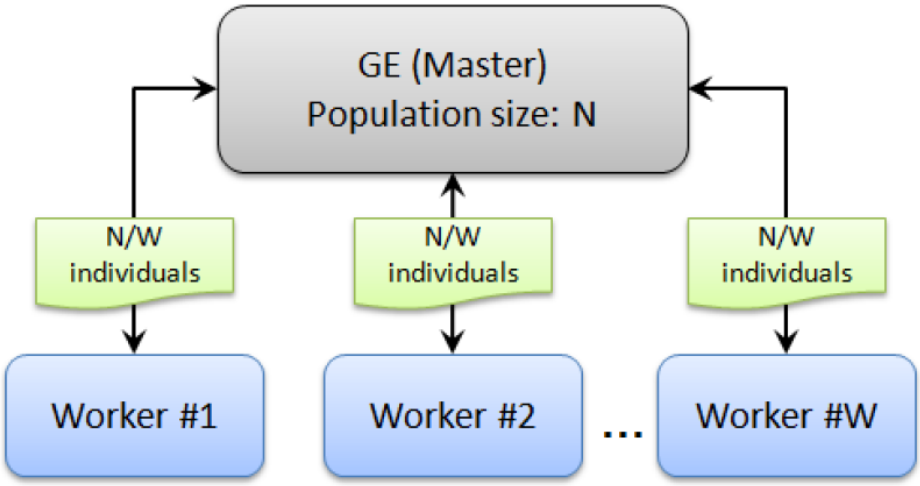


**Figure 29. Master-worker scheme**

As in the serial algorithm, selection and recombination are global: each individual may compete and recombinate with any other. The evaluation of the individuals is usually parallelized, because the fitness of an individual is independent from the rest of the population and there is no need to communicate during this phase. Communication occurs only as each worker receives its subset of individuals to evaluate and when the workers return the fitness values. This parallel algorithm is synchronous because the algorithm stops and waits the fitness values for all the population before proceeding into the next generation. As a result, synchronous algorithms have the same properties as the sequential ones, but it is faster if the algorithm spends most of the time for the evaluation process as is the case. Synchronous master-worker schemes have many advantages: they explore the search space exactly as a sequential algorithm, they are easy to implement and significant performance improvements are possible in many cases [44].

Figure 29 shows the parallel process. There is one master node and W workers. The master node executes all the GE operations except the fitness evaluation. In the evaluation step,

a set of N/W individuals are sent to every worker, where N is the total population size. Next, the master waits for the results computed by the workers. So, in the best case, if the algorithm spends most of the time for the evaluation process, the total execution time is reduced in a T/W factor, where T is the execution time of the serial algorithm. Note that in the optimization of embedded systems described in this research work, the evaluation of the different DMMs generated is about 98% of the entire computational time. The algorithm shown in Figure 29 follows a distributed or multi-threaded design, which is suitable to be executed in multi-core architectures, as well as workstations connected over a LAN. The approach here implemented consists of executing the proposed pGE in a set of PCs connected over a LAN. To this end, there have been used two workstations of two cores each where have been executed up to 1 master and 3 workers using two workstations of two cores each, through the DEVS/SOA framework proposed in [45]. DEVS is a general formalism for discrete event system modeling based on set theory [46]. Once a system is described in terms of the DEVS theory, it can be easily implemented using an existing library. DEVS/SOA is a new DEVS simulation framework based on web services called Discrete Event Systems Specification over Service Oriented Architecture. The main advantage of using DEVS/SOA is that the original model may be distributed with no additional parallelization middleware support, i.e., the whole system can be distributed using a standard DEVS library. Another major advantage is that DEVS/SOA allows the engineer to combine several DEVS platforms to model a system, i.e., it provides interoperability between multiple (and distributed) processing architectures. Thus, each workstation executes two threads. Individuals are sent from master to workers (located at different workstations) and vice versa using web services. In the next section, this process is explained in detail using two real-life embedded multimedia applications.

## 4.3 Experimental Results

The proposed methodology has been applied to two case studies that represent different modern multimedia application domains. The first case study is a 3D Physics Engine (Physics3D) for elastic and deformable bodies [41], which displays the 3D interaction of non-rigid bodies. The second benchmark is VDrift [42], which is a driving simulation game. The game includes as main features: 19 different tracks, 28 types of cars, artificial intelligent players and a networked multi-player mode. To compare the results with a well-known DMM, a

implementation of one of the fastest general-purpose DM managers has been done using the DMM library, namely the Kingsley memory allocator (described in Chapter 2.1) [7].

The parameters employed in the GE algorithm for both applications are shown in Table 3. To implement the GE algorithm, as in the previous chapter GEVA [47] has been used, a well-known GE tool written in Java. The distributed version is made by adding the DEVS/SOA framework proposed in [45], which will utilize multiple processors when available. The experiments have been made using two Intel Core 2 CPU 6600 2.40GHz workstations with 2GB DDR memory each one.

| Parameter | Value |
|---|---|
| **Population Size** | 60 |
| **Number of Generations** | 100 |
| **Probability of Crossover** | 0.80 |
| **Probability of Mutation** | 0.02 |

**Table 3. pGE algorithm parameters**

### 4.3.1 Quality of solutions

In this section are shown the results obtained in both sequential and parallel implementations. Both algorithms are running the same number of generations. As a consequence of the master-worker scheme, the results obtained for performance, memory usage and energy consumption are quite similar in both architectures, i.e., the best DMMs generated using GE or pGE are almost identical.

### 4.3.2 Method applied to Physics3D

To create the final custom DMM, the proposed methodology flow explained in Chapter 3 has been followed. The first step is to profile the behavior of the application using a basic DMM implementation. Then, the Grammar Filter tool is run with the following results: first, it makes the decision to have many block sizes to prevent internal fragmentation. This is done because

the memory blocks requested by the Physics3D application vary greatly in size (to store bodies of different sizes) and if only one block size is used for all the different block sizes requested, the internal fragmentation would be large. Next, the tool chooses that splitting or coalescing must be available for the DMM implementations, so that every time a memory block with a bigger or smaller size than the current block is requested, the splitting and coalescing mechanisms are invoked. At the end of this phase, a reduced grammar file is obtained, which is used by both GE and pGE algorithm. Then, the optimization phase is run and a comparison is done among the custom solutions with the Kingsley [7] DM manager.

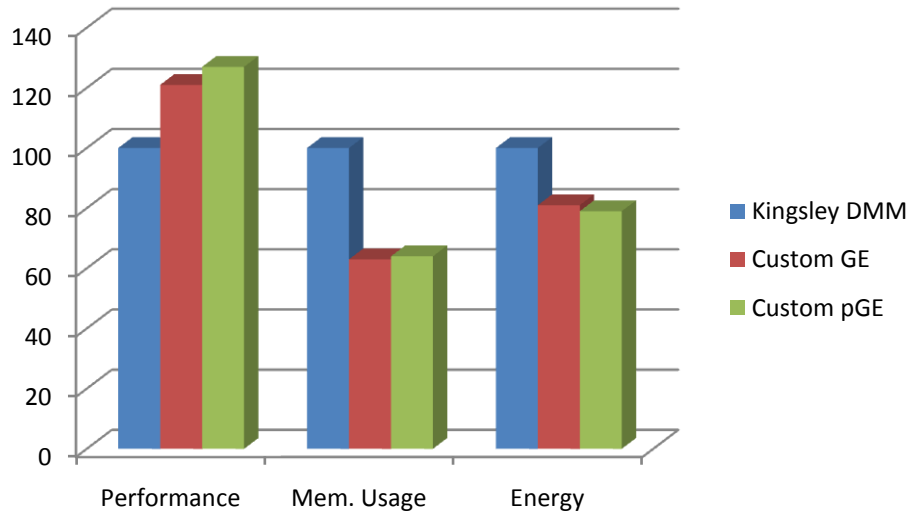


**Figure 30. Physics3D – Results Parallelization**

Figure 30 depicts the best results obtained by both the serial and parallel implementation of the algorithm (using 3 workers), denoted as "Custom - GE" and "Custom - pGE", respectively, and normalized to Kingsley. As it can be seen, the results obtained by GE and pGE are quite similar. However, since the number of generations is the same and pGE is much faster, the parallelization of the algorithm is able to improve the custom DMM results due to a wider exploration of candidate solutions when the optimization time is set to the same value. As Figure 30 shows, the custom DMMs obtained with the parallel implementation use less memory (reduction of 37% in GE and 36% in pGE) and energy (19% and 21% less) than Kingsley. This is due to the fact that the custom DMM managers do not have fixed sized blocks to try with multiple accesses, and try to coalesce and split as much as needed to efficiently use the existing memory, which is a better option in dynamic applications with large variations in requested sizes. Moreover, when large coalesced chunks of memory are not used, they are returned back

to the system for other applications. Furthermore, the results indicate that the custom DMMs achieve significantly better results for performance (21% and 27% increase in GE and pGE, respectively), when compared to Kingsley, because most of the dynamic accesses performed internally by Kingsley to its complex management structures are not required in the custom DMMs, which use a simpler and optimized internal data structures for the target application.

Even though Kingsley does not perform splitting or coalescing operations, it suffers from a large memory footprint penalty and performs unnecessary accesses to traverse all its storage bins in order to find the closest size for each new requested memory allocation. This translates into many unnecessary accesses (and expensive ones, because bigger memories need to be used) with respect to the custom DMMs. Consequently, for Physics3D, the presented methodology allows to design a much customized DMMs that exhibit less fragmentation than Kingsley and, thus, require less memory. Moreover, since this decrease in memory usage is combined with a simpler internal management of DM, the final DM managers need less energy and obtain significant improvements in performance as well.

### 4.3.3 Method applied to VDrift

As it was explained in the previous chapter, the dynamic behavior of the VDrift case study shows that only a very limited range of data type sizes are used in it, namely 11 different allocation sizes are requested. In addition, most of these allocated sizes are relatively small (i.e., between 32 or 8192 Bytes) and only very few blocks are much bigger (e.g., 151 Kbytes). Furthermore, it is possible to see that most of the data types interact with each other and are alive almost all the execution time of the application. Within this context, the methodology exposed is applied. In this case both the GE and pGE algorithms offer the same custom DMM. As a result, the final solution obtained consists of a custom DMM with 4 separated pools or regions for the relevant sizes in the application (in both GE and pGE best solutions). The first pool is used for the smallest allocation size requested in the application, that is, 32 bytes. The second pool allows allocations of sizes between 756 bytes and 1024 bytes. Then, the third pool is used for allocation requests of 8192 bytes. Finally, the fourth pool is used for big allocation requests blocks (e.g., 151 or 265 Kbytes). The pool for the smallest size has its blocks in a single-linked list because it does not need to coalesce or split since only one block size can be requested in it. The rest of the pools include doubly linked lists of free blocks with headers that contain the size of each respective block and information about their current state (i.e., in use or free). These mechanisms efficiently support immediate coalescing and splitting inside these pools, which minimizes both internal and external fragmentation in the custom DMM designed

with the methodology proposed. The DMM Kingsley defined and used in previous sections has been also used here to compare the results. The performance, memory used and energy consumed by the best DMMs obtained using GE and pGE are depicted in Figure 31 and compared to Kingsley.

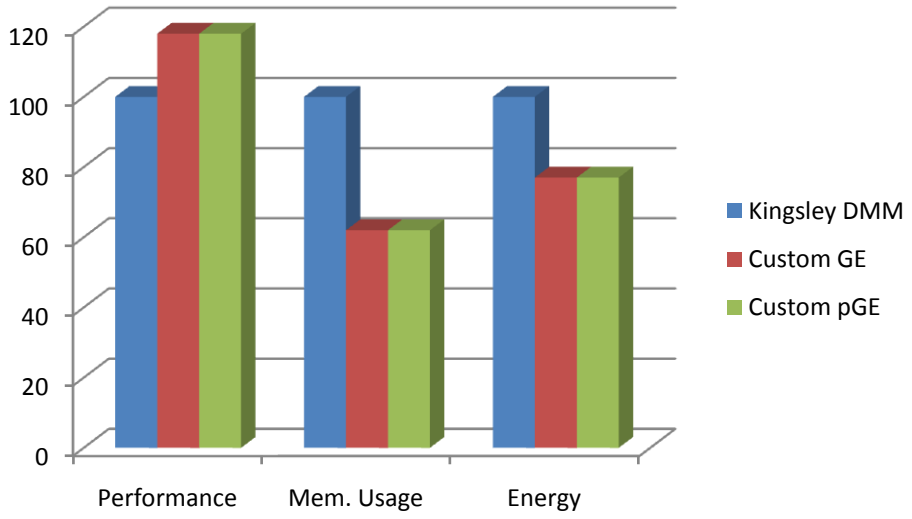


**Figure 31. VDrift – Results Parallelization**

In this case, as the parallel implementation of the optimization algorithm obtains the same DMM that the normal implementation of the optimization algorithm, the results obtained here are similar to the results obtained in the Chapter three. These results show that the values obtained with the DMM designed using the proposed methodology obtains significant improvements in memory usage compared to the manually designed implementation of Kingsley (38%). This result is obtained because our custom DMM is able to minimize the fragmentation of the system in two ways. First, because its design and behavior varies according to the different block sizes requested. Second, in pools where a range of block sizes requests are allowed, it uses immediate coalescing and splitting services to reduce both internal and external fragmentation. In Kingsley an initial boundary memory is reserved and distributed among the different lists for sizes. In this case, since only a limited amount of sizes is used, some of the "bins" (or pools of DM blocks in Kingsley) [15] are underused. Therefore, the custom DMMs consume less energy than Kingsley (i.e., a reduction of 23%). In addition, the final embedded system implementation using the custom DMMs achieve better performance results than the implementations using the Kingsley DMM (18% of improvement).

### 4.3.4 GE and pGE exploration speeds comparison

Finally, a comparison of the performance results for the different sequential and parallel exploration algorithms is shown in Figure 32. To this end, a comparison of the execution time employed to explore optimal DMMs using both the sequential and the parallel GE is done. Moreover, three different parallel configurations have been tested with 1, 2, and 3 workers.

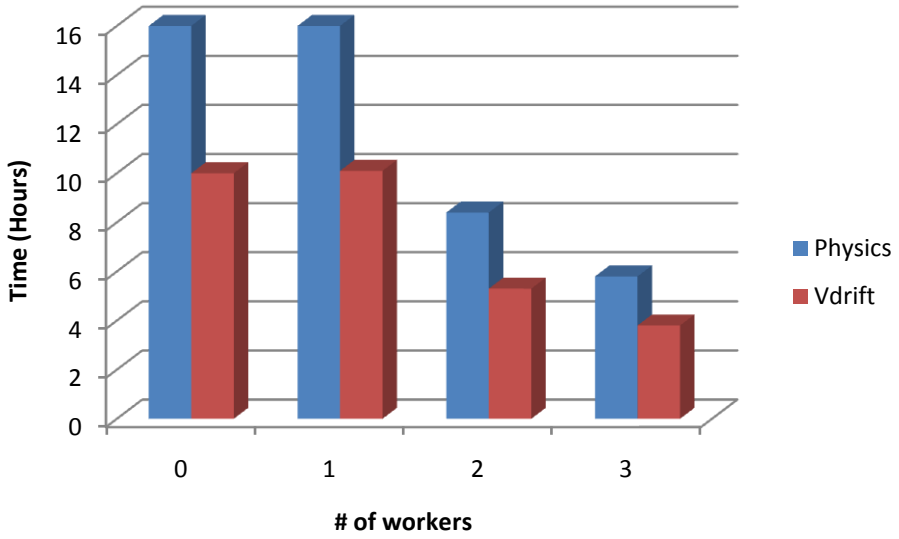


**Figure 32. Results for the exploration speed of pGE algorithms**

As Figure 32 depicts, the slowest algorithm is pGE with 1 worker, reaching speed-ups of 0.98 and 0.97 for Physics3D and VDrift, respectively. Note that in this case, there are two nodes, the first one is a master node running the GE algorithm and the other one is performing the evaluation of the entire population, sequentially. It means that there is not parallelism here. As a result, because of the communication time, the pGE algorithm has the highest execution time in both Physics and VDrift applications. The fastest algorithm is pGE with 3 workers, reaching speed-ups of 2.78 and 2.69 with respect to the sequential GE for Physics3D and VDrift, respectively. As a result, the pGE scales almost linearly with respect to the number of workers used. Obviously, the higher the number of workers, the larger the communication time, so the final gain in the parallelization is being decreased because of the communication process. In this case, the penalization is high because of the use of web services as the communication protocol, which spends more time in the initialization of both client and server than in the individuals passing process.

## 4.4 Conclusions

In this chapter a study about the parallelization of our methodologies has been show.

In our DMM design flow the GE algorithm has a great computational effort (simulate the behavior of each of the DMMs in the application), therefore it is a candidate to be parallelized. We have shown how to parallelize the GE algorithm that evaluate and create new DMMs.

We have followed the methodology obtaining a DMM. And we have compared the results of this DMM with the results of a DMM obtained without parallelization and with the Kingsley Allocator. We can conclude that the results of our DMMs (with and without parallelization) are quite similar, but always better than the Kingsley results. Furthermore the DMM obtained by the parallelized flow is obtained at least four times faster than by the approach without parallelization.

# 5. TOWARD MEMORY RELIABILITY IN DMMs OPTIMIZATION

Embedded computing systems have become a pervasive part of daily life, used for tasks ranging from providing entertainment to assisting the functioning of key human organs.

While mission-critical embedded applications raise obvious reliability concerns, unexpected or premature failures in even noncritical applications such as game boxes and portable video players can erode a manufacturer's reputation and greatly diminish widespread acceptability of new devices. The advent of more sophisticated embedded systems that support more powerful functions, and the reliance on deep submicron process technologies for their fabrication, have brought reliability concerns to the forefront [48].

## 5.1 Reliability Issues

Reliability is an old concern that has been already pursued in high-end military and aerospace markets. Three major trends can be found to affect system reliability according to different time-span intervals [49], [48].

- The first trend is coming from silicon manufacturing process variations due to the statistical nature of these processes, which result in a large percentage of

produced devices to operate below the minimum acceptable speed. This variability, both inter-die but especially intra-die, has an unpredictable impact right after manufacturing on the main performance and energy metrics of devices and interconnects [50]. Several works has been done in this area, trying to minimize the impact of this variability in the delay of the system using different memory architectures [51] or using configurable memories [52].

- The second trend is the transient faults [53] that are errors caused by temporary conditions on the chip (such as when power supply noise or interconnect noise exceed a certain threshold) or by external noise (such as soft errors caused by neutrons striking the chip). The circuit itself is not damaged even though computational errors are introduced. In this case, circuit- and system-level techniques can reduce these transient errors (e.g., redundant execution with voting schemes, double sampling flip-flops, etc.) [54].
- The third trend refers to non-reversible failures in advanced periods of system lifetime due to thermal effects and aging of the devices. In this case, both HW and software approaches have been proposed [49].

In this chapter we have done a study to know if while we are designing DMMs it is possible to reduce the probability of the aging effect without having an impact in the main metrics for embedded systems: performance, energy consumption and memory use.

## 5.2 Aging Effect

Embedded system designs using new process technologies cause higher on-chip temperatures, which result from higher power densities. This fact significantly accelerates various failure mechanisms, leading to an overall decrease in reliability. Accelerated aging creates a serious risk that devices will fail within an embedded system's warranty period, which poses the key challenge of finding countermeasures that effectively prolong a system's lifetime. One of the failure mechanisms is the Soft- Breakdown (SBD), it appears when enough traps align in the gate dielectric and then a conducting path is created resulting in "micro" tunneling currents through the gate. After some time the path created will "burn out" leading to a short or Hard Break-down (HBD) [55] resulting in a catastrophic failure. The transition from the initial conducting path to the HBD is not abrupt, the gate current will start progressively increasing

long before the HBD occurs (see Figure 33**¡Error! No se encuentra el origen de la referencia.**).

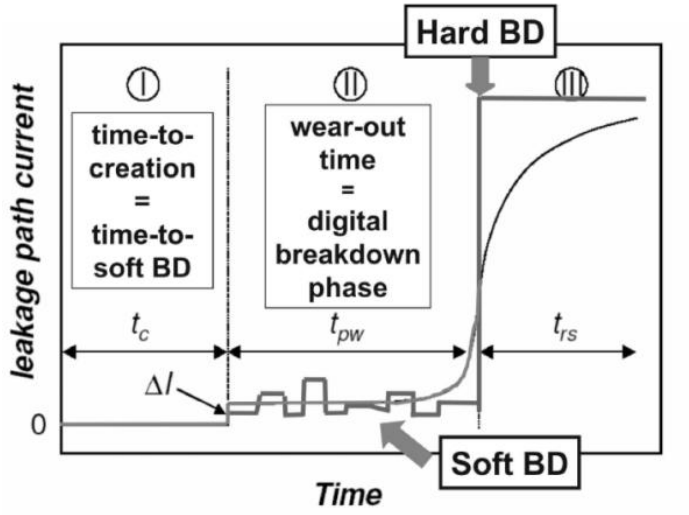


**Figure 33. Breakdown Model for a gate**

After a certain time using the gate (stress time) it will enter in the SBD state. In this state the gate continue working, but it doesn't have a normal functionality, usually the delay of the gate increase and the quantity is not predictable. The effect of a SBD is to increase the margins of functionality in the energy and in the delay. This effect will be added to the variation in the process effect, and the final result is a really high variability to take in account (see Figure 34). If the stress continues in the gate during a certain time, it will finish in a hard breakdown, after that this cell cannot work anymore. But if when the gate is in the soft breakdown state, the stress stops during a while, the gate could return to the initial state and to the normal functionality.

The goal of this research is to obtain optimized DMMs but in this case, more reliable. The way to improve the reliability in this research is traverse the aging effect, therefore what is needed is to reduce the stress time of the gates.

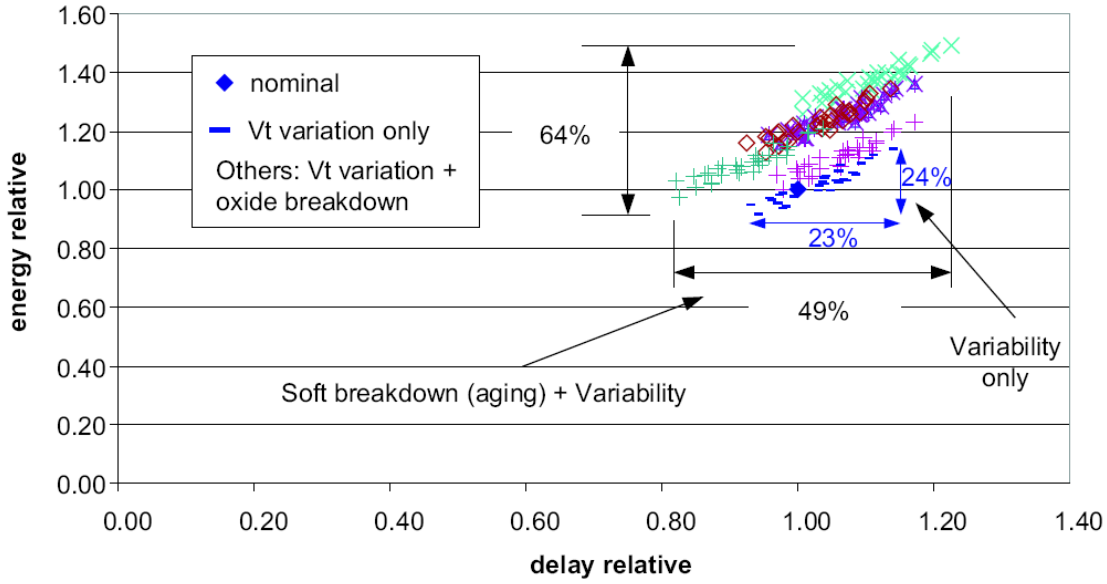


**Figure 34. Impact of variability and SBD in the energy delay**

## 5.3 Method Applied

In this research work is shown a preventive approach to the aging effect. As it has been said, to reduce the aging effect it is important to reduce the stress time of the gates. In addition, if a reduction of the power consumption is got, then it also helps to reduce the aging effect because of the fact that the heat dissipation will be less (the heat dissipation is one of the causes of the aging effect). It is really complex to monitor all the gates of the system. Instead of this, in this research work the path followed is to divide the memory in "virtual" regions. These regions do not correspond to physical divisions in the memory. The idea behind this proposition is that if the number of consecutive accesses done to a determinate region is reduced, then consequently the time that a gate of that region is in use will be also reduced. This is the same that reduce the stress time and therefore we will reduce the probability of the appearance of the aging effect.

It is proposed a system of management that is composed by a certain number of DMMs (one to manage each of the different regions), and in addition is needed a mechanism to decide during execution time what is the region in which one block must be allocated. The mechanism selected act as a LRU policy that means that the next region to allocate a block is the one that was used the longest time ago. This policy has an overhead mainly in the performance and in

the energy consumption, which is bigger as big is the set of regions to manage. For this reason it is not recommended to divide the memory in a big number of regions as it will be shown.

To obtain the system of management, the optimization flow proposed in the Chapter 3 is modified to create a system of DMMs instead a unique DMM. In the next section the new flow is explained.

## 5.4 DMM Modified Optimization Flow

The final objective of this chapter is to obtain a system of management that includes optimal DMMs maintaining a reliable memory system, to that end we have done several modifications to the design flow presented in the Chapter 3. The proposed optimization framework, as the previous one, uses three different phases to perform the automatic exploration to create the management system using Grammatical Evolution (GE). Modifications have been done in the three phases. Figure 35 shows the new optimization flow. In first place there is the profiling phase, but in this case it creates several profiling. In the second step, the grammar filter phase, several grammars are formed, one for each of the profiling reports. And finally the optimization phase obtains as a final result a set of optimal DMMs. The modifications are described in the next paragraphs.

In the profiling phase, as in the previous proposed flow, the application under study is run using a basic DMM implementation. In this design flow the number of regions of memory desired must be introduced as a parameter. With this information and with the one extracted from the application, the algorithm will create n profiling reports, where n is the number of regions defined by the designer. The LRU policy has been used to distribute the instructions among all the profiling reports. The goal using this policy is to disperse the accesses to the memory with the objective of reducing the consecutive accesses to any of the regions in the memory. Therefore at the end of this phase there are n profiling reports

The next phase, the grammar filter phase, is like it was in the previous defined optimization flow. But in this case it has the special feature that is that instead of create one grammar, it create n grammars, one specific for each of the profiling reports.

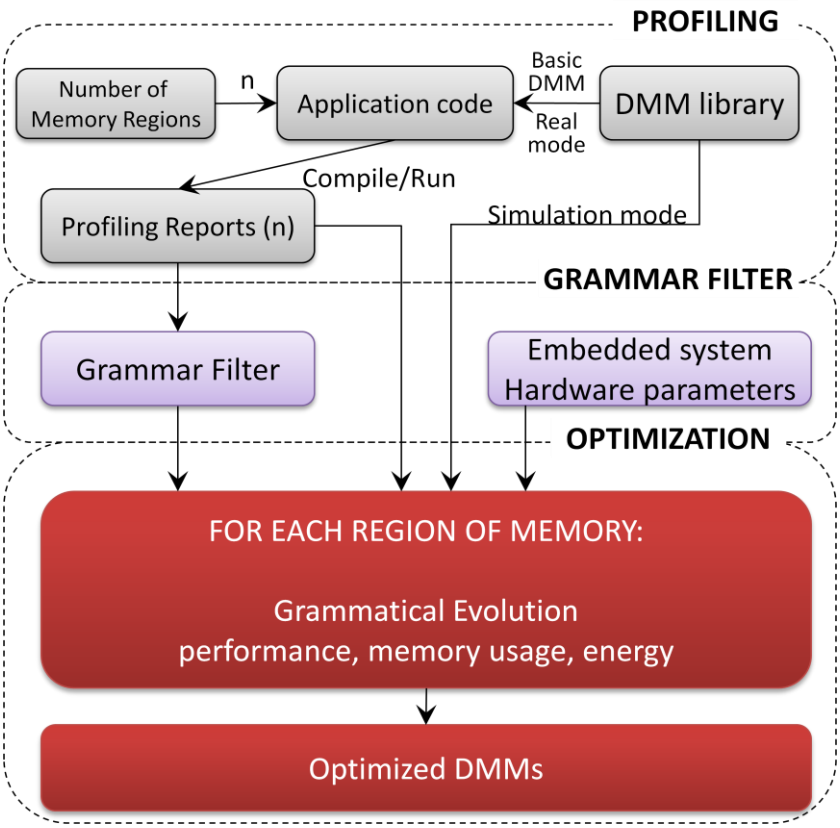


**Figure 35. Reliability Management System Optimization Flow**

As in the old optimization flow, the last phase is the optimization process. The inputs now are the n profiling reports, the n grammars and the embedded hardware parameters. The simulation mode of the DMM library has not been modified. It works as before obtaining the fitness of a determinate DMM as the balanced sum of their performance, energy consumption and memory usage. The optimization algorithm is the same that the previous defined but now it is done n times, one for each of the defined memory regions. And each of the times done, it create a DMM specific for one profiling report (Figure 36). The final result of this phase is a set of n optimal DMMs, one for each of the defined regions.

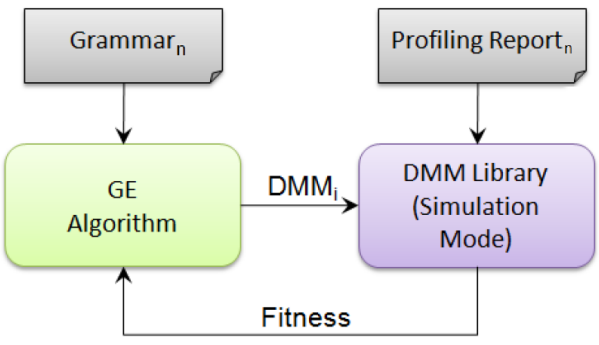


**Figure 36. Optimization Algorithm for each memory region**

The final result of the complete DMM optimization flow is the system of management, composed by the set of n optimal DMMs and the mechanism to decide to what region each block must go.

## 5.5 Experimental Results

The proposed methodology, as in previous chapters (Chapter 3 and Chapter 4) has been applied to two case studies that represent different modern multimedia application domains: the first case study is a 3D Physics Engine (Physics3D) for elastic and deformable bodies [41], which is a 3D engine that displays the interaction of non-rigid bodies. The second benchmark is VDrift [42], which is a driving simulation game. The game includes as main features: 19 different tracks, 28 types of cars, artificial intelligent players and a networked multi-player mode. The parameters employed in the GE algorithm for both applications are shown in Table 4. The obtained results have been compared with the results obtained by the old methodology (described in the Chapter 3).

| Parameter | Value |
|---|---|
| **Population Size** | 50 |
| **Number of Generations** | 100 |
| **Probability of crossover** | 0.80 |
| **Probability of mutation** | 0.02 |

**Table 4. Parameters for the GE algorithm with reliability support**

To study the reliability of the system an analysis of two metrics has been done, in one hand the arithmetic mean of all the consecutive accesses done is computed. This metric is important to see if the final system of management makes as a general a lot of consecutive accesses (less reliable) or not (more reliable). It also gives the recover capacity of a region. A low number of this metric means that if a high number of consecutive accesses exist then probably the next numbers of consecutive accesses will be low. During the time that the consecutive accesses are no intensive, the gates came back to the initial state and therefore the

risk of aging effect decreases. In the other hand it is the maximum number of consecutive accesses done during the application execution. As bigger this number is, bigger is the chance of get the HBD state. So if the maximum number of consecutive accesses is decreased, the memory will be more reliable because the stress time will be fewer. This research is working at the level of memory regions, so to make the comparison between this new reliable approach and the previous non reliable approach, it is necessary to suppose a "virtual" division of memory that the previous approach uses in regions. To that end and to make a fair comparison, the memory used by the original approach is divided in the same divisions that are considering in the reliable implementation, i.e. the memory of the previous implementation does not change, but we have added an extension that count the accesses done to each of the false divisions to make the final comparison.

### 5.5.1 Method applied to Physics3D

As we divide the memory in regions, it is necessary to use a structure to manage the accesses to each region, i.e. the LRU list. This mechanism adds some overhead to the final system that must be computed. For this end in the Figure 37 and Figure 38 we show the extra overhead added to the final system in terms of computational effort (Figure 37) and energy consumption (Figure 38). The third relevant metric for the embedded systems, the memory usage, is not shown because the increase in memory usage is not significant (less than 20% for 32 regions). In the figures the one region configuration is not shown because it does not need a mechanism to manage the accesses to different regions and therefore it does not have extra overhead. The results shown in the figures are normalized to the two regions configuration.

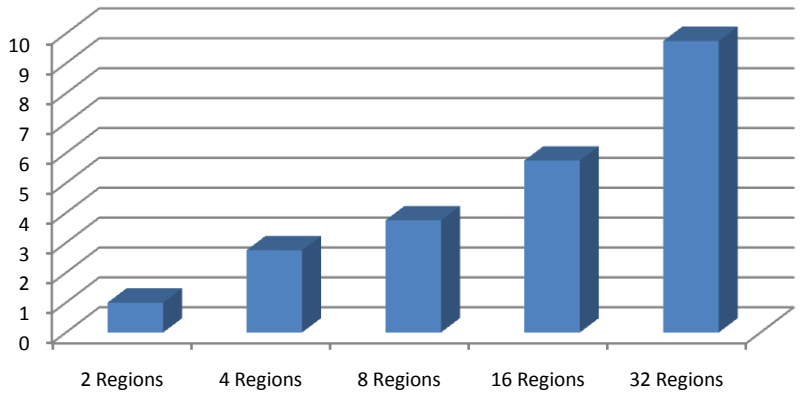


**Figure 37. Physics3D – Extra computational effort per memory configuration (%)**

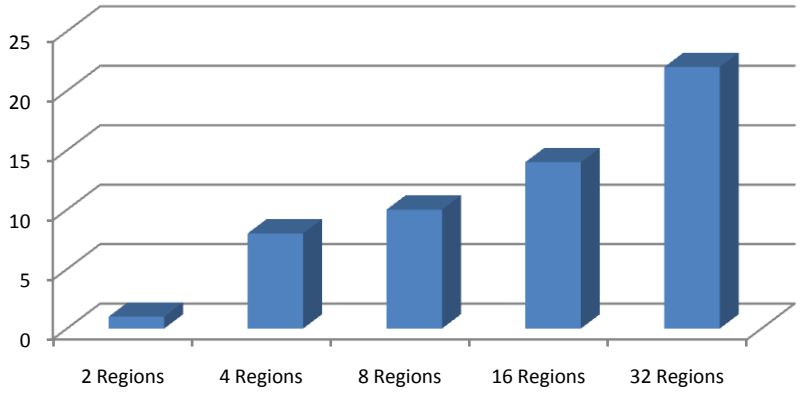


**Figure 38. Physics3D – Extra energy consumption per memory configuration (%)**

As we can see, the configuration that adds less overhead is the two regions configuration. The big step from the two to four regions memory configurations is due that the LRU policy is really simple to manage two regions, but as we increase the number of regions, the management is more and more complex. This fact is reflected in the use of memory and in the computational effort. Note that in the energy consumption the bad result of increase the number of regions is more aggressive than for the computational effort. This appreciation is important because energy consumption is one of the sources of the aging effect (due to the heat dissipation) and we are looking to reduce it. Therefore is not compatible to reduce the aging effect at the same time that the heat dissipation is increased.

In addition to these figures a comparison between the maximum number of consecutive accesses to a same region doing in each of the memory configurations is shown (Figure 39).

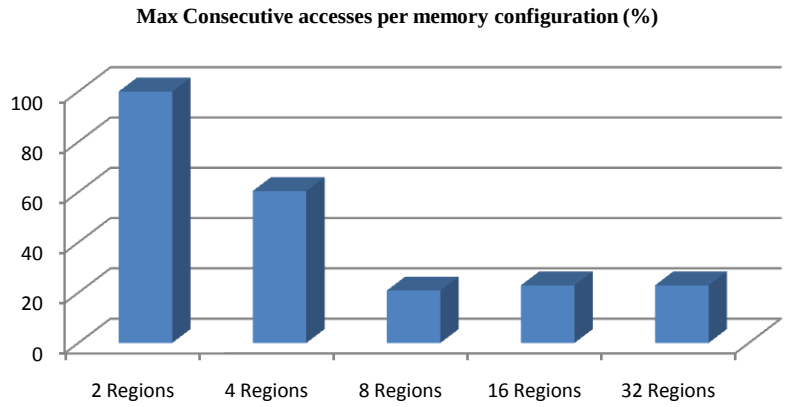


**Figure 39. Physics3D – Max consecutive accesses per memory configuration (%)**

In this case the maximum number of consecutive errors produced in the two division memory configuration is almost double than in the four division memory configuration. The eight regions memory configuration is still better but more than eight regions do not improve in terms of maximum number of consecutive accesses.

We can see that there are two confront results, in one side with fewer regions it obtains less overhead, but in the other side with fewer regions it obtains a bigger number for the maximum consecutive accesses. It is necessary to make a trade-off between these metrics but there is an important constraint, it is not desirable to deteriorate the performance, energy consumption and memory usage obtained without the reliable extension. As the objective of this section of this research work is to obtain a reliable system, increase the power consumption is not an option, so these facts point to the two division memory configuration as the best configuration. But to reassert this, all the experiments will be done with two and four regions configuration to see if the decision taken is a good decision.

To create the reliable system of management, the proposed methodology has been followed. The first step is the profile of the behavior of the application using a basic DMM implementation, in this case it is necessary to pass as parameter two regions memory configuration (four regions memory configuration). Then two (four) profiling reports are obtained in this phase, one for each of the regions of memory. Then for each of these regions the optimal DMM is obtained. Therefore at the end of the algorithm there is a reliable system of management formed by two (four) different DMMs that manage one memory region each and the mechanism to decide the location of the blocks (add some overhead to the final system).

First of all there is a consecutive accesses study (Figure 40 and Figure 41). As explained before, the information shown is the arithmetic mean of the consecutive accesses and the

maximum number of consecutive accesses done during the application execution compared to the original approach for the two configurations in study.

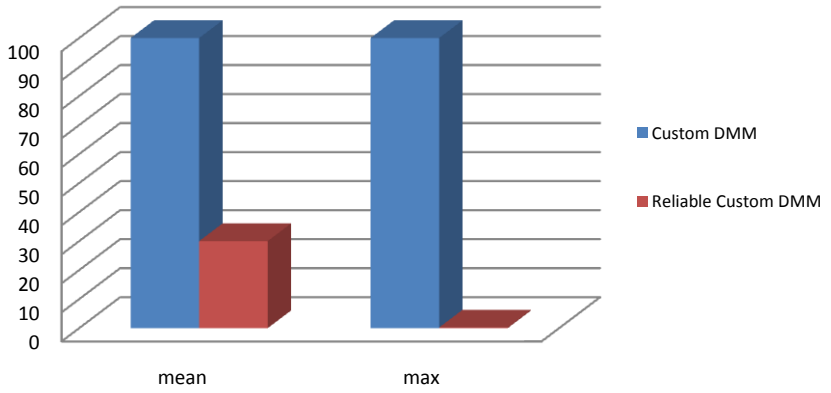


**Figure 40. Physics3D – Two regions memory configuration consecutive accesses study (%)**

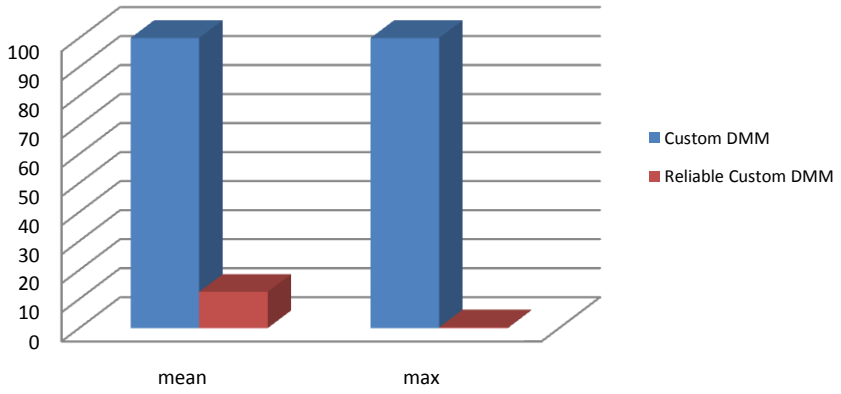


**Figure 41. Physics3D – Four regions memory configuration consecutive accesses study (%)**

Regarding the arithmetic mean of the number of the consecutive accesses it is possible to see that as expected both of the configurations obtain an important reduction. In the case of the four regions configuration the decrease (87%) is bigger than for the two regions configuration (70%). Both of them reduce the arithmetic mean what means that in general the number of consecutive accesses done is less than for the other implementation, reducing in this way, and in mean, the stress time induced to each region. Looking at the maximum number of consecutive accesses, the reduction in percentage is quite similar (more than 99.7% for both of them) because although in absolute dates the maximum number of consecutive accesses of the two regions configuration doubles the maximum number of consecutive accesses of the four regions

configuration, the number of consecutive accesses of the previous implementation is so high that any of the implementations has a significant reduction. The reduction of the maximum number of consecutive accesses also reduces the stress time induced to the region and therefore reduces the probability of aging effect.

As the final result of the application of our methodology, we obtain a system of management form by the combination of two (four) DMMs plus the mechanism that implement the LRU policy. In the next figures (Figure 42 and Figure 43) we show the comparison between our results and the results obtained by the previous presented approach in the main metrics, power consumption, performance and memory usage for the two different configurations.

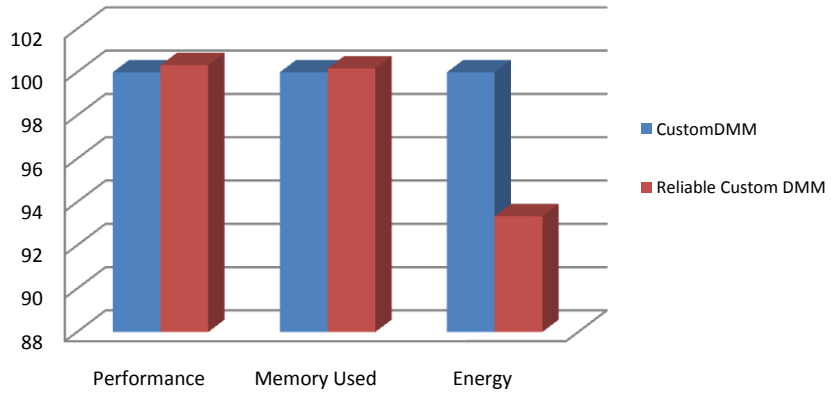


**Figure 42. Physics3D – Two regions memory configuration results (%)**

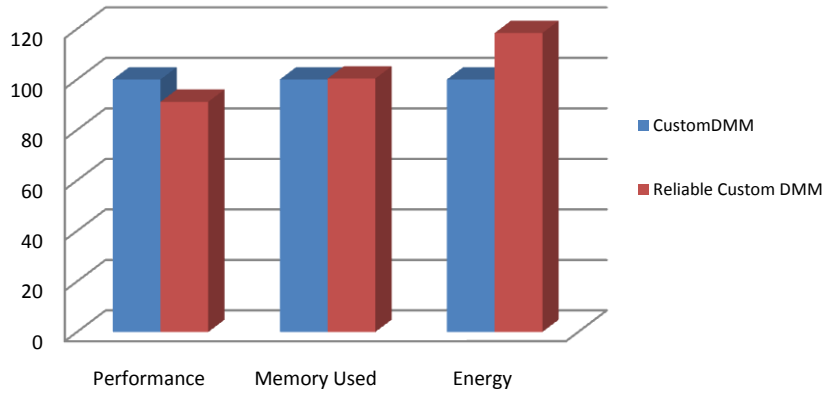


**Figure 43. Physics3D – Four regions memory configuration results (%)**

At this point and looking at the results, we can conclude that the use of a four regions memory configuration (and therefore all the memory configuration with a bigger memory division) must not be used in this study. The reason is that despite of having a desired behavior with the consecutive accesses to the memory, the overhead introduced by the mechanism that implements the LRU policy is too elevated. A decrease in performance (- 9%) is achieved. This point is not a big problem because it is possible to let a little decrease in the performance if a more reliable system is obtained. But the big problem arises at the energy consumption. In this case, the increment (18%) in the energy consumption has also effect in the aging effect due to the heat dissipation. Therefore the increment in the energy consumption is not permitted for the objective of increase the reliability. With these experiments is proved that for divisions of more than two regions, the extra overhead does not compensate the gain in the number of consecutive memory accesses to the same region.

Regarding to the two regions memory configuration, it maintains the performance (actually is increased in 0.3%) obtained by the approach without the reliability. This fact is due that in one side is obtained a increase of the performance because of the fact of manage blocks in different regions of memory with shorter lists (less complex to manage) and in the other side there is a performance penalty because of the extra overhead added by the LRU mechanism. Concerning to the memory usage, it is almost the same (there is an almost 0.2% more memory used) that the approach without reliability. This fact is due to the extra memory needed to the LRU mechanism and the needed memory for the new DMMs. And finally, in the case of the energy consumption is obtained a reduction (- 6.5%) in contrast to the approach without reliability. This reduction is really important and it has the double effect: in one hand it also helps to reduce the chance of HBD reducing the heat dissipation, and in the other hand the less energy consumption let to produce smaller devices and with more autonomy.

### 5.5.2 Method applied to VDrift

As in the previous application, the first thing to do is to see a study that reaffirms the conclusion that states that the two regions configuration is the best for the purpose of this research work. To that end Figure 44 and Figure 45 show the extra overhead added when we use the LRU mechanism to classify the instructions in regions. Figure 44 depicts the extra overhead added to the final memory system by each of the memory configuration to the computational effort. Figure 45 depicts the extra overhead added to the final memory system by each of the memory configuration to the power consumption. The third relevant metric for the embedded systems,

the memory usage, is not shown because the increase in memory usage is not significant (less than 15% for 32 regions).

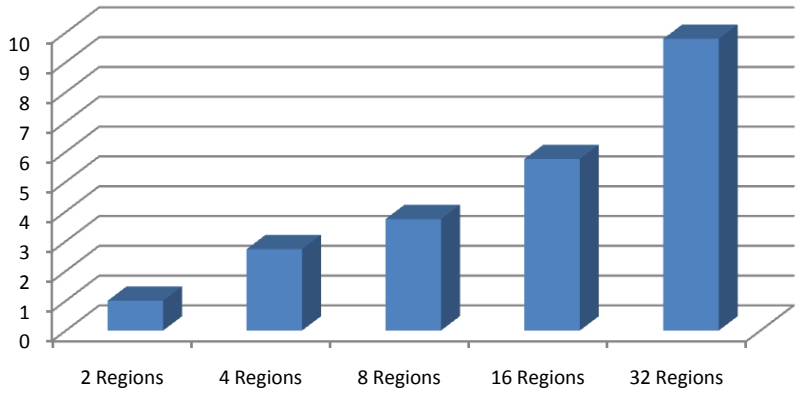


**Figure 44. VDrift – Extra computational effort per memory configuration (%)**

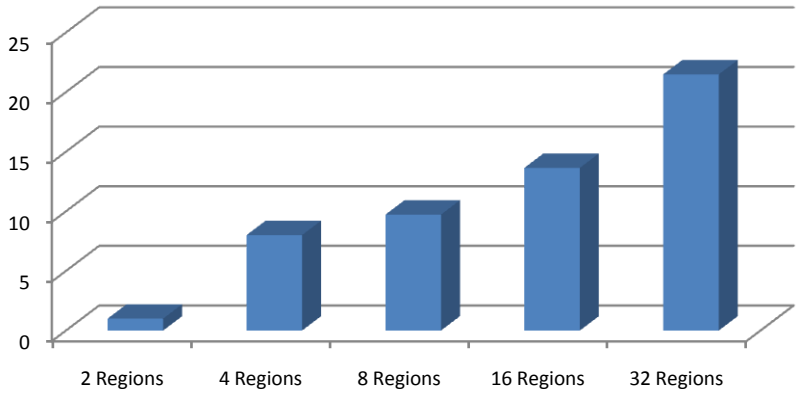


**Figure 45. VDrift – Extra energy consumption per memory configuration (%)**

As in Physics3D, the best configuration in terms of extra overhead added is the two regions memory configuration. The results obtained here are quite similar to the physics3D results, the two metrics increase in the extra overhead as more memory regions are added because of the complexity of manage with the LRU policy big lists. And also the difference in overhead between the two and four regions memory configurations is quite big (more than two times for the computational effort and more than seven times for the energy consumption).

To complete the analysis that guarantees the decision of take as the best configuration the two regions memory configuration, as for the previous application, Figure 46 depicts the maximum number of consecutive accesses for each of the configurations normalized to the two regions configuration.

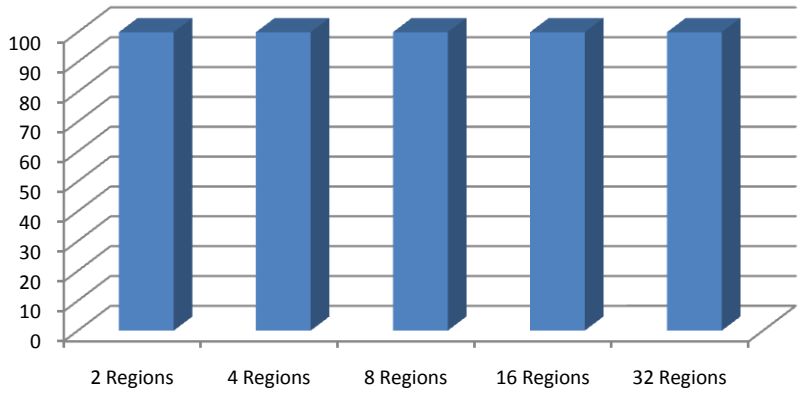


**Figure 46. VDrift – Max Consecutive accesses per memory configuration (%)**

In this case, even when the number of regions of memory is increased, the number of maximum consecutive accesses done does not decrease. There is a basic reason for this behavior, when there are a certain number of consecutive reads and writes to the same block, as this block is in one unique region of memory, and the instructions are consecutive, all the accesses must done to the same region (the region where the block is allocated). Thus, there is an inferior limit for the maximum consecutive accesses that in this case have been reached with the two regions memory configuration. Then for the case of VDrift, the information contained in the previous figures adduced that the configuration that gets the less overhead and at the same time gets the less maximum number of consecutive accesses is the two regions memory configuration. Consequently as we have proved that the best configuration for our purpose is the two regions configuration, for the next results we only show the results obtained by this configuration.

At this point, we apply our methodology. The number of memory regions is two. As result of the design process a system of management is provided. This system is composed by two optimal DMMs one for each of the memory regions and the mechanism to decide the region in which the blocks must be allocated. The results of this implementation are compared to the results obtained by the implementation without the reliability extension.

First of all we show the consecutive accesses study to see if the proposed methodology has a final impact in the reliability of the system. As for physics3D Figure 47 shows the arithmetic mean of the consecutive accesses and the maximum number of consecutive accesses done during the application execution compared to our approach without the reliability extension.

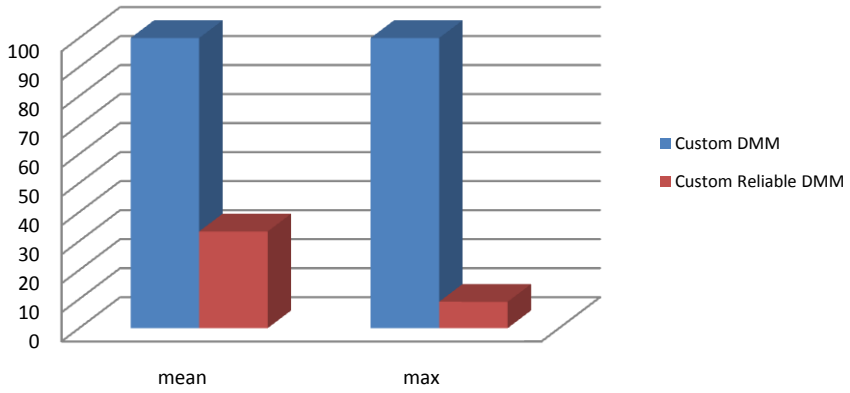


**Figure 47. VDrift – Consecutive accesses study for two regions memory configuration (%)**

The final system of management reduces the arithmetic mean in a 66%. The effect of this is that in general our implementation make a less number of consecutive accesses each time it access to a determinate region than the other implementation. In this way it reduces the probability of obtain a big number of consecutive accesses to any of the regions, and therefore reduces the probability of aging effect. Looking at the maximum number of consecutive accesses done during the program execution, it has been reduced in a 90%. This data means that the maximum stress time induced to a determinate region in one punctual moment has been decreased. Thence by this side we have also reduced the aging effect.

Next, Figure 48 shows the results obtained by our new methodology and the methodology without the reliability extension [56] in the main metrics: performance, energy consumption and memory usage. The results are normalized to the ones obtained by the original methodology.

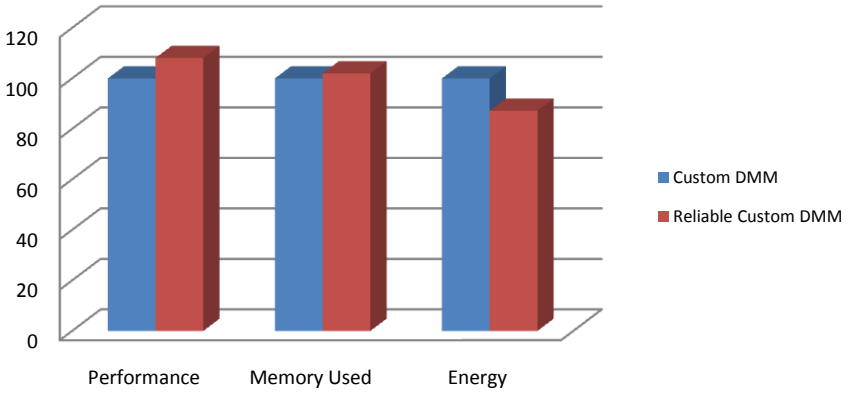


**Figure 48. VDrift – Two regions memory configuration results (%)**

As in physics3D the results are really good. There is an increase in the performance (8%) because fewer instructions must be executed to find a block in a list (because the DMMs manage fewer blocks). The memory used by the system is increased (2%) due to that as now we have two DMMs there is needed extra memory to support it. And regarding the energy consumption, we obtain a reduction (7%) that helps to decrease the aging effect, because of the accelerating aging effect due to the heat dissipation. This reduction is obtained because of the better distribution of the free blocks in the DMMs that let to reduce the number of memory accesses and hence the energy consumption.

## 5.4 Conclusions

Reliability is a concern that each day is more important due to the scaling technologies and because of the consequences that errors have in the final systems.

In this chapter we have done an initial study about the possibility of increase the reliability of the systems from our methodologies to create DMMs. To this end we have modified our previous DMMs design flow.

We have afforded this problem dividing the memory in regions and creating different DMMs to manage each of these regions. The main problem is that we need mechanisms to decide in runtime in what region a block must be allocated. These mechanisms add extra overhead to the final system and make unaffordable several memory divisions.

Our experimental results provide the ideal configuration to obtain a more reliable system maintaining the important metrics in the embedded systems: performance, energy consumption and memory used. This configuration is the division of memory in two regions because the overhead added to the system is not relevant and the number of consecutive accesses is decreased. In this way the probability of aging effect is reduced.

Furthermore we have shown that the results in the main metrics for the two regions configuration are better than the results obtained by our methodology without the reliability extension. We have increased the performance at least 0.3% and we have decreased the energy consumption at least 6%. The reduction in the energy consumption has an added effect to the reliability. When we reduce the energy consumption we also reduce the heat dissipation, and hence the aging effect.

# 6. CONCLUSIONS AND FUTURE WORK

Nowadays, high-performance embedded devices (e.g., PDAs, advanced mobile phones, portable video games stations, etc.) need to execute very dynamic embedded applications (3D games, video or music players, etc.). These applications are complex and typically they are written in high-level languages (C++ or Java). These applications have the special feature that they demand intensive dynamic memory due to the unpredictability of the volume of input data as of the user behavior. Therefore there is a necessity to optimize the way they access to the memory system with the end of offer good features to the final users (performance, battery life and number of concurrent applications). This optimization must be afforded from the layer of the OS that attend the requests of dynamic memory from the application, which is the dynamic memory manager.

In previous research works, these requirements have been afforded with a methodology that explores the design space of DMMs and creates a final optimized DMM for the application in study. This methodology try in one side to optimize the final DMM and in the other side to reduce the time needed to create a new DMM following that methodology. Unfortunately within this context, the manual exploration and optimization of the DMM implementation is one of the most time-consuming and programming-intensive parts.

In this research work is presented new system-level approach that redefines previous DMM design methodologies. In this case is shown a novel DMM optimization flow based on grammatical evolution to automatically characterize custom DMMs with an integrated profiling method. This approach largely simplifies the complex engineering process of designing and

profiling several implementation candidates, allowing the developers to automatically cover a vast part of the DM management design space (e.g., different strategies of the heap, internal blocks of the allocators, etc.) without any programming and modeling effort. In addition to obtain a bigger reduction in the design time needed to create new DMMs, a parallel implementation for the methodology has been presented. Finally due to the scaling technologies, the reliability of these systems has become an important factor. In this research work is also included a study on how improve the reliability of the system from the perspective of the DMMs. As it has been shown in the case studies, the results obtained for the main metrics in the embedded systems (performance, memory usage and power consumption) by the obtained optimized DMM using GE are significantly better than those obtained by other previous similar approaches or by a well-know general-purpose allocators. And in addition the time needed to obtain the DMM has been reduced and the reliability of the system has been enhanced.

## 6.1 Conclusions

Along the development of this research work, some conclusions have arisen and some of them have been reflected in the chapters of this work. In this section a summary of them is presented:

a) The behavior of the dynamic applications executed in the embedded systems is unpredictable. Because of this reason a DMM must exist to control all the request of DM. To that end, recent works have developed methodologies to create specific DMMs for each system. But these methodologies require a high interaction of the designer that must take several decisions in the design process. Moreover the process is timing-consuming because of the exploration of the design space and the need of evaluate each of the DMM alternatives.

b) The state-of-the-art methodologies do not take in account reliability issues that due to the scaling technologies are each day more present in the embedded systems.

c) To afford these inefficiencies in this research work is presented a novel methodology to create optimal DMMs. This methodology is based in GE algorithms, which make the process automatic, avoiding the interaction of the designer and reducing the time of the entire process. Our experimental results show that using our methodologies it is possible to obtain improvements in performance of 56%, reductions in memory used of 20% and reductions in energy consumption of 70% in contrast to general purpose approaches.

d) The parallelization of the methodology proposed (mainly the GE algorithm) reduces still more the time needed to create custom DMMs, maintaining the final results in the important metrics. The time needed to create a new DMM is reduced four times in contrast to the methodology without the parallelization.

e) To afford the reliability problems, the proposed flow is modified. It includes the necessary mechanisms to reduce the chance of the appearance of the aging effect that is one of the reliability concerns. In addition to the reduction of the aging effect the final results are improved. The performance is increased in at least 0.3% and the energy consumption is reduced in at least 7%.

## 6.2 Main Contributions

As a summary, the main contributions of this research work have been:

a) The implementation of a tool that given a profiling report of an application, it creates a grammar. This grammar defines all the possible DMMs that can be constructing. Every possible DMM formed will be adapted to the application behavior thanks to this tool (Chapter 3).

b) The presentation of a novel DMM design flow based in some concepts of previous methodologies. The revolutionary idea behind this methodology is the use of GE algorithms to explore the grammar that cover the DMM design space. The main important features of this DMM design flow are that the flow is automatic (it means that there is no need of the designer interaction) and besides it completes the process of obtain the optimal DMM in a reduced time because of the addition of a simulation mode that let to evaluate automatically all the DMMs proposed by the algorithm in a short period of time (Chapter 3).

c) The parallelization of the DMM design flow. This modification makes the proposed methodology faster because the evaluation of the different DMMs can be done automatically at the same time, reducing in this way the final time spent in the process (Chapter 4).

d) A study about adding reliability to the memory modifying the design flow. A mechanism to reduce the chance of the aging effect has been added to the algorithm.

Additionally the new way of manage the regions of memory let the final system to obtain better results in the main metrics (Chapter 4).

## 6.3 Future work

There are some open paths that can be afforded after this research work:

a) Multi-Objective Optimization. In this research work has been used a basic multi-objective optimization approach. There are three objectives that require an optimization: performance, memory usage and energy consumption. In this work they have been optimized using a weight sum that for the proposed goals is enough. But if a Multi-Objective EA (MOEA) is used, it could improve further the quality of the solutions in the exploration process and it will avoid the action of decide the weight for each of the metrics. There is a wide research in the MOEAs that can be explored for this purpose.

b) Multi-processor environments. This study has been done supposing monoprocessor systems or multiprocessor system but without data sharing between tasks. The multiprocessor systems are each day more common, and they have a great potential to reduce the energy consumption and to increase the performance. But there are some new problems that arise with these systems. As an example, one of them is the possibility of sharing resources. Most of the new features that arise with the multiprocessors are not cover by the current methodology and it could be modified to cover them.

c) Addition of extra data structures. The current design space includes lists as mechanism to storage the blocks that are free. There are some other data structures, like the trees that could be useful in some situations. To add new structures some modifications must be done, first of all in the design space and after that in the block header because they will need more information. These modifications can affect to the memory usage or even to the performance so it is necessary to make an impact study before.

d) Dynamic Adaptation. This research work has presented a methodology that as a final result obtains a DMM. This DMM is specific for the system and for a representative set of the inputs to the system. It is probably that in the near future, the number of the

applications and the unpredictability of the inputs increase. For this reason a study of adaptable DMMs could be done. These managers must adapt their functionality and internal structure to the workload present in the system in each moment.

e) Reliability issues. In this an initial study has been presented. This study can be extended trying to reduce the overhead when more than two regions are defined. In addition there are more trends in reliability like transient errors or variability that could be tackling in some way. Concerning the variability, the previous proposed line (dynamic adaptation) could be used to change the features of the DMM in run time depending on the position inside the margins of the variability where the system is.

# 7. REFERENCES


[1] T. Weise, *Global Optimization Algorithms - Theory and Application* -, 2nd Edition ed.: http://www.it-weise.de, 2008.

[2] G. E. Moore, "Cramming more components into integrating circuits," *Electronics,* vol. 38, 1965.

[3] D. Atienza, "Metodología multinivel de refinamiento del subsistema de memoria dinámca para los sistemas empotrados de altas prestaciones," Tesis Doctoral, Universidad Complutense de Madrid, 2005.

[4] J. L. Risco-Martín, D. Atienza, J. I. Hidalgo *et al.*, "A parallel evolutionary algorithm to optimize dynamic data types in embedded systems.," *Soft Comput.*, vol. 12, no. 12, pp. 1157-1167, 2008.

[5] S. M. D. Atienza, F. Catthoor, J. M. Mendias, and D. Soudris, "Dynamic memory management design methodology for reduced memory footprint in multimedia and wireless network applications," in DATE: Proceedings of the conference on Design, automation and test in Europe, Washington, DC, USA, 2004.

[6] N. Vijaykrishnan, M. Kandemir, M. J. Irwin *et al.*, "Evaluating integrated hardware-software optimizations using a unied energy estimation framework," *IEEE Trans. Comput*, vol. 52, no. 1, pp. 59-76, 2003.

[7] P. Wilson, M. Johnstone, M. Neely *et al.*, "Dynamic Storage Allocation: A Survey and Critical Review," in Proceedings of the Workshop on Memory Management, New York, 1995.

[8] A. L.Arboníes, *Nuevos enfoques en la innovación de productos para la empresa industrial*: Díaz de Santos, 1993.

[9] D. Atienza, S. Mamagkakis, F. Poletti *et al.*, "Efficient system-level prototyping of power-aware dynamic memory managers for embedded systems," *Integration, the VLSI Journal*, vol. 39, no. 2, pp. 113 - 130, 2006.

[10] D. Atienza, J. M. Mendias, S. Mamagkakis *et al.*, “Systematic dynamic memory management design methodology for reduced memory footprint,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 11, no. 2, pp. 465 - 489, 2006.

[11] E. D. Berger, B. G. Zorn, and K. S. McKinley, “Composing high-performance memory allocators,” in Proceedings of Conference PLDI, New York, 2001.

[12] S. Mamagkakis, D. Atienza, C. Poucet *et al*, “Energy- Efficient Dynamic Memory Allocators at the Middleware Level of Embedded Systems,” in EMSOFT, Seoul, Korea, 2006.

[13] M. MSDN. "Windows Embedded CE 6.0 Advanced Memory Management."

[14] D. Grunwald, and B. Zorn, “CustoMalloc: efficient synthesized memory allocators,” *Software—Practice & Experience*, vol. 23, no. 8, pp. 851 - 869 1993.

[15] D. A. Barrett, and B. G. Zorn, “Using lifetime predictors to improve memory allocation performance,” in Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, Albuquerque, New Mexico, United States, 1993, pp. 187-196.

[16] K.-P. Vo, “Vmalloc : A general and efficient memory allocator,” *Software Practice & Experience*, vol. 26, pp. 1-18, 1996.

[17] D. Atienza, S. Mamagkakis, F. Catthoor *et al.*, “Reducing memory accesses with a system-level design methodology in customized dynamic memory management,” *2nd Workshop on Embedded Systems for Real-Time Multimedia*, pp. 93- 98, 2004.

[18] S. Mamagkakis, D. Atienza, C. Poucet *et al.*, "Energy-Efficient Dynamic Memory Allocators at the Middleware Level of Embedded Systems." pp. 215-222.

[19] E. Daylight, D. Atienza, A. Vandecappelle *et al.*, “Memory access-aware data structure transformations for embedded software with dynamic data accesses.,” in IEEE Transactions, 2004.

[20] Stroustrup, *The C++ programmming Language*, Boston, USA: Addison-Wesley Longman Publishing Co. Inc., 2000.

[21] I. Das, and J. Dennis, “A closer look at drawbacks of minimizing weighted sums of objectives for Pareto set generation in multicriteria optimization problems,” 1997.

[22] C. C. Coello, and G. Lamont, *Applications of Multi-Objective Evolutionary Algorithms*: World Scientific, 2004.

[23] K. Deb, *Multiobjective Optimization Using Evolutionary Algorithms*: John Wiley and Son Ltd., 2001.

[24] C. Darwin, *On the origin of Species*: John Murray, 1859.

[25] G. Mendel, “Experiments in Plant Hybridization,” in meetings of the Brünn Natural History Society, 1865.

[26] T. Bäck, U. Hammel, and H.-P. Schwefel, “Evolutionary computation: comments on the history and current state.,” *IEEE Transactions on Evolutionary Computation*, 1997.

[27] J. E. Baker, “Reducing bias and inefficiency in the selection algorithm,” in Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application, Cambridge, Massachusetts, United States, 1987.

[28] A. E. Eiben, and J. E. Smith, *Introduction to Evolutionary Computing*: Springer, Natural Computing Series, 2007.

[29] A. S. Fraser, “Simulation of Genetic Systems by Automatic Digital Computers,” *Australian Journal of Biological Science*, 1957.

[30] J. H. Holland., *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.*, 1975.

[31] I. Rechenberg, *Cybernetic Solution Path of an Experimental Problem. Royal Air-craft Establishment*: Library Translation 1122, 1965.

[32] L. J. Fogel, “On the organization of intellect,” UCLA University of California, Los Angeles, California, USA, 1964.

[33] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection.*: The MIT Press, 1992.

[34] C. Ryan , J. J. Collins and M. O'Neill, "Grammatical Evolution: Evolving Programs for an Arbitrary Language."

[35] A. Brabazon, M. O'Neill and I. Dempsey “An Introduction to Evolutionary Computation in Finance,” *Computational Intelligence Magazine, IEEE,* vol. 3, no. 4, pp. 42-55, Nov. 2008.

[36] I. Dempsey, M. O'Neill, and A. Brabazon, “Constant creation in grammatical evolution,” *International Journal of Innovative Computing and Applications,* vol. 1, no. 1, pp. 23-38, 2007

[37] C. Ryan, “Grammatical Evolution Tutorial,” in Proceedings of Genetic and Evolutionary Computation Conference, GECCO 2006.

[38] M. Mamidipaka, and N. Dutt, *eCACTI: An Enhanced Power Estimation Model for On-chip Caches*, Center for Embedded Computer Systems (CECS), 2004.

[39] M. Sipser, *Introduction to the Theory of Computation*: Course Technology, 2005.

[40] J. I. Hidalgo, J. L. Risco-Martín, D. Atienza *et al.*, "Analysis of multi-objective evolutionary algorithms to optimize dynamic data types in embedded systems." pp. 1515-1522,.

[41] L. Kharevych, and R. Khan, "3D physics engine for elastic and deformable bodies," University of Maryland, College Park, 2002.

[42] "VDrift racing simulator," http://vdrift.net, 2008.

[43] E. Cantú-Paz, “A Survey of Parallel Genetic Algorithms,” *Calculateurs Paralleles,* vol. 10, 1998.

[44] E. Cantú-Paz, "Designing Efficient Master-Slave Parallel Genetic Algorithms." pp. 455–460.

[45] S. Mittal, J. L. Risco-Martín, and B. P. Zeigler, “DEVS/SOA: A Cross-Platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process,” *SIMULATION: Transactions of SCS,* vol. 85, no. 7, pp. 419-450, 2009.

[46] B. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, 2000.

[47] M. O'Neill, E. Hemberg, C. Gilligan *et al.*, "Geva - grammatical evolution in java. ," Technical report, Natural Computing Research & Applications Group - UCD Complex & Adaptive Systems Laboratory, University College Dublin, Ireland, 2008.

[48] V. Narayanan, and Y. Xie, "Reliability concerns in embedded system designs," *Computer,* vol. 39, no. 1, pp. 118- 120, 2006.

[49] D. Atienza , G. D. Micheli, Luca Benini *et al*, "Reliability-Aware Design for Nanometer-Scale Devices."

[50] A. Papanikolaou, H. Wang, M. Miranda *et al.*, "Reliability Issues in Deep Deep Submicron Technologies: Time-Dependent Variability and its Impact on Embedded System Design," in Proceedings of the 13th IEEE International On-Line Testing Symposium, 2007 pp. 121.

[51] A. Papanikolaou, T. Grabner, M. Miranda *et al.*, "Yield prediction for architecture exploration in nanometer technology nodes:: a model and case study for memory organizations," in Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. , Seoul, 2006.

[52] C. Sanz, M. Prieto, J. I. Gómez *et al.*, "System-level process variability compensation on memory organizations: on the scalability of multi-mode memories," in Proceedings of the 2009 Asia and South Pacific Design Automation Conference, Yokohama, Japan, 2009 pp. 254-259

[53] R. Baumann, "Soft Errors in Advanced Computer Systems," *IEEE Design & Test,* vol. 22, no. 3, pp. 258 - 266, 2005.

[54] S. Mitra, N. Seifert, M. Zhang *et al.*, "Robust System Design with Built-In Soft-Error Resilience," *Computer,* vol. 38, no. 2, pp. 43 - 52 2005.

[55] J. S. Suehle, B. Zhu, Y. Chen *et al.*, "Detailed study and projection of hard breakdown evolution in ultra-thin gate oxides," *Microelectronics and reliability,* vol. 45, no. 3-4, pp. 419-426, 2005.

# APPENDIX 1
# PUBLICATIONS

- José L. Risco-Martín, David Atienza, Rubén Gonzalo and J. Ignacio Hidalgo. "Optimization of dynamic memory managers for embedded systems using grammatical evolution" in GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, Montreal, 2009.

  [Based on its impact factor, GECCO is 11th in the rankings of 701 international conferences in artificial intelligence, machine learning, robotics, and human-computer interaction.]

- José L. Risco-Martín, José M. Colmenar and Rubén Gonzalo. "A parallel evolutionary algorithm to optimize dynamic memory managers in embedded systems". In WPABA'10: Proceedings of the Third International Workshop on Parallel Architectures and Bioinspired Algorithms at the NineTeenth International Conference on Parallel Architectures and Compilation Techniques PABA, 2010

  [This paper has been sent and accepted]

# APPENDIX 2
# ABBREVIATURES

| | |
|---|---|
| **cDMMs** | Custom Dynamic Memory Managers |
| **DEVS** | Discrete Event Specification |
| **DM** | Dynamic Memory |
| **DMM** | Dynamic Memory Manager |
| **EA** | Evolutionary Algorithms |
| **EP** | Evolutionary Programming |
| **ES** | Evolution Strategy |
| **GA** | Genetic Algorithms |
| **GE** | Grammatical Evolution |
| **GP** | Genetic Programming |
| **gpm** | genoty-phenotype mapping |
| **MOEA** | Multi Objective Evolutionary Algorithms |
| **MOOP** | Multi Objective Optimization Problems |
| **MPSoC** | Multi Processor System on Chip |
| **OS** | Operating System |
| **SOA** | Service Oriented Architecture |
| **SoC** | System on Chip |

# APPENDIX 3
# TABLE OF FIGURES